I. INTRODUCTION

Even after years of progress both in simulation-based and formal verification techniques, System-on-Chip (SoC) verification at the Register-Transfer-Level (RTL) has remained one of the main bottlenecks in the overall SoC design flow. While more and more sub-systems of increasing size become integrated on a single chip the level of abstraction for state-of-the-art verification techniques has remained unchanged. Even with the advent of new design methodologies based on Electronic System Level (ESL) descriptions, it is still the RTL that has remained the point of reference for creating the “golden” design model.

A comparison with verification practices at lower design levels may illustrate the problem: as a matter of course, when verifying the logic input/output behavior of a large combinational circuit, we consider its gate-level description. Simulating its transistor-level description instead, in principle, is possible but would be considered highly inefficient and inappropriate. Similarly, when verifying the micro-architecture of an SoC design, we base this verification on its RTL description rather than its gate netlist. This is standard procedure, because we can trust the gate netlist and the RTL description: they are considered sound abstractions for the next lower levels.

By contrast, when evaluating the global system behavior of an SoC, this verification is done at the RTL. Verification of high-level models can be useful to build confidence in the specification at an early design phase, but it is not adequate to render a full-blown, system-wide RTL verification superfluous. We only trust the actual implementation of our SoC after chip-level simulations on the RTL have been running for weeks and months on large workstation clusters. Why do we not rely on verifying the global system behavior at the system level? Obviously, we do not trust our system-level models as much as we trust our RTL models and our gate models. The reason for this lack of confidence is well known and has often been addressed as the “semantic gap” between the system and the RT level. In fact, establishing a well-defined, formal relationship between these two levels is a difficult problem. It is one of the main hurdles when attempting to integrate system-level models into standard SoC design and verification flows.

This paper intends to contribute a theoretical basis and the sketch of a methodology to create sound abstractions of RTL descriptions at the system level. We envision that, based on the proposed methodology, system-level verification techniques, e.g., based on SystemC [1], [2], [3], [4], [5], [6], in the future, can be used to replace chip-level verification at the RTL.

In the field of automated theorem proving it has already been a paradigm for a long time to create a stack of sound models to support system verification by abstract models [7]. On the other hand, theorem proving requires a lot of expertise and manual effort by the verification engineer. The approach proposed here has drawn some inspiration from the results of [8], [9], [10], where concepts like stuttering bisimulation have been used to bridge the semantic gap between different abstraction levels. However, it is the objective of this paper to create a formal link between system models and the concrete RTL implementation only by employing standard techniques of property checking, as they are already common practice in industry.

The basic idea of this work is to close the gap between the system level and the RTL by defining the semantics of the
system model through the systematic formulation of properties in a standard property language like SystemVerilog Assertions (SVA) [11] or Property Specification Language (PSL) [12]. If the required properties are proven, the system model is a sound abstraction of the RTL implementation.

The authors are not aware of many other attempts to use standard property checking for this purpose. A notable exception is [13]. Their work describes an approach from mathematical logic based on interpretation of theories to relate time-abstract system descriptions to cycle-accurate implementations. In contrast, our approach [14] is based on the conventional formalisms of model checking such as Kripke structures. Moreover, the theoretical framework proposed here extends over [13] and our previous work [14] in that it is fully compositional. Our approach can handle the concurrency between RTL modules that are each represented by time-abstract models at the system level [15].

A different approach to reduce the verification costs in system-level design flows is to employ high-level synthesis and/or high-level equivalence checking [16], [17]. High-level equivalence checking is very appropriate in applications where a notion of equivalence can be established between the system level and the RTL. This is quite often the case in data path applications. However, in other applications the notion of equivalence can be too strict to describe the relationship between the abstract and the concrete model. In [10], [18], [19] equivalence checking is combined with theorem proving to obtain a more general framework for linking concrete RTL implementations with high level models. In general, when both the concrete design and the abstract model are available, such approaches to high-level equivalence checking may nicely complement the envisioned design and verification flow proposed here.

The paper is organized as follows: Path predicate abstraction is introduced based on the notion of operational graph coloring in Sec. II. This already provides an intuition on the nature of the proposed abstraction for SoC verification. Sec. III explains how property checking can be used to create path predicate abstractions for finite state machines (FSMs). The soundness of the abstract models with respect to the concrete FSMs is established in Sec. IV. Sec. V extends the proposed formalisms to make path predicate abstraction compositional and states a soundness theorem for composed systems. In Sec. VI it is illustrated by means of an example how our methodology can be implemented in practice. Two industrial case studies have been conducted to evaluate the proposed methodology. This is subject of Sec. VII.

II. PATH PREDICATE ABSTRACTION IN DIRECTED GRAPHS

The main objective of this paper is to show that standard property checking techniques, when applied in a systematic way, can be used to create sound abstractions of RTL designs. The notion of an “operation” plays a key role in our methodology. We first introduce path predicate abstraction for directed graphs based on a graph coloring function. This already provides an intuition how operations can be used to create sound abstractions. Later the graph notions of this section will be extended to finite state machines (FSMs) and property checking in Kripke models.

**Definition 1 [Operational Graph Coloring]:**
Consider a directed graph \( G = (V,E) \), a subset \( W \subseteq V \) of the graph vertices called colored nodes, a set of colors \( \hat{W} = \{ \hat{w}_1, \hat{w}_2, \ldots \} \) and a surjective coloring function \( c : W \rightarrow \hat{W} \). A path \( (v_0, v_1, \ldots, v_n) \) such that \( v_0, v_n \in W \) and \( v_1, \ldots, v_{n-1} \in V \setminus W \) is called operational path in \( G \). The set \( W \) must be chosen and colored such that:

1) every cyclic path in \( G \) contains at least one node from \( W \) (no cycles with only uncolored nodes in the graph),
2) for every operational path \( (v_0, v_1, \ldots, v_n) \) and \( u_0 \in W \) such that \( c(u_0) = c(v_0) \) there must exist an operational path \( (u_0, u_1, \ldots, u_m) \) in \( G \) with \( c(u_m) = c(v_n) \)

We call \( c \) an operational coloring function and \( G \) an operationally colored graph. □

![Fig. 1: Example of an operational coloring](image1)

Fig. 1 shows an example of an operationally colored graph. The blue (b), green (g) and yellow (y) vertices are elements of \( W \subseteq V \). The white nodes do not belong to \( W \). As can be seen, the first condition of Def. 1 is fulfilled as there is no cyclic path along only white nodes. Also the second condition can be checked easily. For example, from every green node there are operational paths to blue nodes as well as to yellow nodes, but there are no operational paths to any other color. Intuitively, the graph describes the “operations”: green→yellow, green→blue, blue→green, yellow→blue.

![Fig. 2: Path predicate abstraction for the operational coloring in Fig 1](image2)

For the operationally colored graph of Fig. 1, we can draw an abstract graph as shown in Fig. 2. It has only a single node for each color. Every operational path in the original graph is represented by an edge in the abstract graph. Such an abstraction is called path predicate abstraction.

**Definition 2 [Path Predicate Abstraction]:**
We consider a graph \( G = (V,E) \) with a set of colored nodes
A directed graph $G = (W, E)$, such that for any two nodes $u, w \in W$, it is $(c(u), c(w)) \in E$ if and only if there is an operational path $(u, \ldots, w)$ in $G$, is called path predicate abstraction of $G$.

In order to understand the role of operations in path predicate abstraction we consider color sequences in the graphs.

**Definition 3** [Color Sequence] Consider a graph $G = (V, E)$ with a set of colored nodes $W \subseteq V$, a set of colors $\hat{W}$ and an operational coloring function $c : W \mapsto \hat{W}$. A **color sequence** produced on a path $(\hat{v}_0, \hat{v}_1, \hat{v}_2, \ldots)$ in $G$ is the sequence of colors ($\hat{w}_0, \hat{w}_1, \ldots$) of the colored nodes on the path, i.e., the $i$-th color in the sequence is $\hat{w}_i = c(s_i)$ if and only $s_i \in W$ is the $i$-th colored node on the path. (Note: it is $i \leq j$ since uncolored nodes $s \not\in W$ on the path do not contribute to the color sequence.)

The following theorem connects color sequences (i.e., sets of paths in the concrete graph) with paths in the abstract graph.

**Theorem 1** [Soundness w.r.t. color sequences]: Let $G = (V, E)$ be a graph and $\hat{G}$ be the path predicate abstraction induced by an operational graph coloring $c : W \mapsto \hat{W}$ with a set of colored nodes $W \subseteq V$.

1. Given an arbitrary finite (infinite) concrete path $(v_0, v_1, v_2, \ldots)$ in $G$ with $v_0 \in V$. There exists a finite (infinite) abstract path $(\hat{w}_0, \hat{w}_1, \ldots)$ in $\hat{G}$ that represents the color sequence produced on the concrete path.

2. Given an arbitrary finite (infinite) abstract path $(\hat{w}_0, \hat{w}_1, \ldots)$ in $\hat{G}$. For every node $v_0 \in V$ in $G$ such that $c(v_0) = \hat{w}_0$ there exists a finite (infinite) concrete path $(v_0, v_1, v_2, \ldots)$ such that the color sequence produced on that path is $(v_0, v_1, \ldots)$.

Proof:

(1) follows immediately from the construction: every colored node $v \in W$ is mapped to the color $\hat{w} = c(v)$ by the coloring function.

(2) We consider the edges traversed on the abstract path. According to requirement (2) of Def. 1, for an edge $(\hat{w}_i, \hat{w}_{i+1})$ there exists an operational path from every colored node $v_i$ with $c(v_i) = \hat{w}_i$ to some colored node $v_j$ with $c(v_j) = \hat{w}_{i+1}$ along uncolored nodes $v_{i+1}, \ldots, v_{j-1}$. Hence, for any abstract path we can find a correspondingly colored concrete path by concatenating the concrete path fragments. Again, according to requirement (2) of Def. 1, such a path can be found for every starting node $v_0$ colored with $c(v_0) = \hat{w}_0$.

Theorem states that a path predicate abstraction is sound with respect to sequences of colors on paths. In other words, every chain of operations in the concrete graph has its correspondence in the abstract graph, and vice versa.

It is interesting to compare this with the well-known notions of bisimulation and stuttering bisimulation [7]. They can be applied in the same way to a colored graph $G$ and an abstract graph $\hat{G}$. Importantly, however, the coloring function would need to be defined for every node in $G$, i.e., it is $V = W$. Then, the relation between $G$ and $\hat{G}$ is a bisimulation iff every path in $G$ can be mapped to a path in $\hat{G}$ such that the sequence of colors is the same. In other words, if $G$ does not contain any uncolored nodes, the notion of path predicate abstraction and bisimulation become identical. In general, however, path predicate abstraction describes a weaker relationship than bisimulation.

The example in Figure 3 shows that a path predicate abstraction also is different from the well-known concept of a stuttering bisimulation. In a stuttering bisimulation the bisimulation is weakened by additionally allowing that paths along nodes in $G$ with identical color (stuttering) can be mapped to a single node in $\hat{G}$ with that color.

For obtaining the abstract graph $\hat{G}$ from the concrete graph $G$ in Figure 3 by a stuttering bisimulation instead of path predicate abstraction it would be required to make $V = W$ by assigning one of the colors green, blue, red or yellow to the uncolored node $v$. However, if $v$ is colored

- green, this wrongly creates an edge blue $\rightarrow$ green in $\hat{G}$,
- blue, this wrongly creates an edge green $\rightarrow$ blue in $\hat{G}$,
- red, this wrongly creates an edge red $\rightarrow$ yellow in $\hat{G}$,
- yellow, this wrongly creates an edge yellow $\rightarrow$ red in $\hat{G}$.

The example illustrates that the main difference between a stuttering bisimulation and our notions results from the fact that path predicate abstraction defines a mapping function only for a subset of nodes. This turns out to be important and facilitates its construction for FSMs and Kripke models by property checking, as will be the subject of the following sections.

When comparing with [7], note, that our work is focussed on a different problem formulation. In [7], a concrete and an abstract model are given. The concrete model implements the abstract model if the externally visible behavior of the concrete is a subset of the externally visible behavior of the abstract. Then the existence of a refinement mapping proving this relationship is examined.

In our work, we are only given the concrete graph $G$ and we are searching for the abstract graph $\hat{G}$. We find this abstract graph by constructing a mapping function, which is formulated based on operational graph coloring and which defines the abstract model. In this context, we may observe that trivial coloring functions always exist that assign every node in the graph a different color or that assign the same color to all nodes. (The resulting path predicate abstractions are, of course, meaningless.)

In the literature of process algebra other weak forms of bisimulations have been proposed. In particular, in [20] a weak bisimulation based on silent transitions is introduced to abstract from certain computational steps in a program. Note however, that a hardware designer can only understand the behavior of an RTL design when looking over several clock cycles. While the semantics of a program is usually understood instruction by instruction, the semantics of an RTL design is blurred over several clock cycles. Therefore, paths from which we want to abstract are usually not completely silent and, in our formulation, we need to model some information also along paths through uncolored nodes. As will be shown, this information can be captured in sequence predicates that are rooted in “important states” corresponding to the colored nodes $W$ of the concrete graph $G$. As will be introduced later...
are Boolean functions characterizing a set of input sequences that covers all major aspects of an industry-strength property checking methodology that are relevant for creating useful path predicate abstractions of an RTL design based on standard property languages.

We first introduce some notations based on a deterministic finite state machine $M = (S, I, X, Y, \delta, \lambda)$ with a set of states $S$, a set of initial states $I \subseteq S$, an input alphabet $X$, an output alphabet $Y$, a transition function $\delta : S \times X \mapsto S$ and an output function $\lambda : S \mapsto Y$.

Sets of states can be identified using state predicates. A state predicate, $\eta(s)$, is a Boolean function evaluating to true for a state $s$ iff $s$ is in the set. A state sequence predicate $\sigma(\pi_l)$ of length $l$ is a Boolean function characterizing a set of state sequences. A state sequence predicate $\sigma$ of length $l$ can be applied to a state sequence $\pi_m$ of arbitrary (non-negative) length $m$. If $m > l$ then $\sigma$ is evaluated on the $l$-prefix of $\pi_m$ (beginning with $s_0$). If $m < l$ then $\sigma$ is evaluated on the set of sequences obtained by extending $\pi_m$ with all possible $(l-m)$-suffixes. The result is true if there exists an extended sequence where $\sigma$ evaluates to true, otherwise it is false.

We also define input and output sequences and predicates for them. An input sequence $\rho_l = (x_0, x_1, \ldots, x_{l-1})$ describes $l$ inputs which may be associated with the $l$ transitions between the $l+1$ states in a state sequence. Accordingly, an output sequence $\gamma_l = (y_1, y_2, \ldots, y_l)$ describes $l$ outputs. Input and output sequences may be characterized using I/O sequence predicates which are defined analogously to state sequence predicates. The sequence predicates $\lambda(\rho_l)$ and $\mu(\gamma_l)$ are Boolean functions characterizing a set of input sequences and output sequences, respectively. (As a convention, we use Greek letters for sequences, predicates and abstraction functions and Roman letters for most of the other objects.)

Since sequence predicates return Boolean values, we can apply the usual Boolean operators $\vee$, $\wedge$, $\neg$, $\Rightarrow$ to sequence predicates. The length of the resulting predicate is the maximum length of the operand predicates.

We define the general path predicate ispath($\pi_l$, $\rho_l$):

$$\text{ispath}((s_0, \ldots, s_l), (x_0, \ldots, x_{l-1})) = \bigwedge_{i=1}^l (\delta(s_{i-1}, x_{i-1}) = s_i)$$

This predicate describes a finite unrolling of the FSM transition relation. We also define the output sequence predicate isoutput($\pi_l$, $\gamma_l$):

$$\text{isoutput}((s_0, \ldots, s_l), (y_1, \ldots, y_l)) = \bigwedge_{i=1}^l (\lambda(s_i) = y_i)$$

It characterizes output sequences as computed by the output function for a given state sequence. Together with ispath it can be used to describe valid output sequences of the FSM.

We can shift predicates in time using the $\text{next}$ operator:

$$\text{next}(\sigma_l, n)(s_0, s_1, \ldots, s_{n+l}) = \sigma_l((s_n, s_{n+1}, \ldots, s_{n+l}))$$

The $\text{next}(\sigma_l, n)$ operator syntactically extends the length of the sequence predicate to $(n + l)$ but semantically it merely shifts the starting point of the evaluation of a predicate $\sigma_l$ by $n$ positions.

A. Operations in FSMs

We now introduce notions that will help us to describe an operational coloring of the state transition graph (STG) of a design’s FSM. The basic idea of obtaining such a coloring is that the behavior of a module is completely described by a set of interval properties, also called operation properties. Each operation property describes a piece of design behavior over a number of clock cycles starting and ending in so called important states. The important states are subsets of the state set $S$. They correspond to the colored nodes in the graph formulation of Sec. II.

Definition 4 [Important-state predicate]:
An important-state predicate $\eta_l(s)$ is a predicate evaluating to true for a set of concrete important states $s$ and to false for all other states. A concrete state cannot satisfy more than one important-state predicate. The disjunction of all $\eta_l(s)$ is a state predicate $\Psi(s) = \eta_1(s) \lor \eta_2(s) \lor \ldots$ characterizing the set of all important states.

An operation is a finite sequence of behavior (i.e., states, inputs, outputs) of a module, moving from an important state to another important state and visiting only unimportant states in between. Operations are triggered by input sequences, and they produce output sequences.

Definition 5 [FSM-operational path]:
An FSM-operational path of length $l$ between two important states, $s_0 \in S$ and $s_l \in S$, is a triple, $(\pi_l, \rho_l, \gamma_l)$, of a state sequence $\pi_l = (s_0, s_1, \ldots, s_{l-1}, s_l)$, an input sequence $\rho_l = (x_0, x_1, \ldots, x_{l-1})$ and an output sequence $\gamma_l = (y_1, y_2, \ldots, y_l)$ with $l > 0$ such that

- $\Psi(s_0) = \text{true}$,
- $\Psi(s_1) = \text{true}$,
- if $i > 1$: all intermediate states $s_1, \ldots, s_{i-1}$ are unimportant states, i.e., $\Psi(s_1) = \ldots = \Psi(s_{i-1}) = \text{false}$,
- the input sequence $\rho_1$ drives the FSM along the path $\pi_1$, i.e., it is $\text{ispath}(\pi_1, \rho_1) = \text{true}$,
- the output sequence $\gamma_i$ is produced by the FSM along the path $\pi_i$, i.e., it is $\text{isoutput}(\pi_i, \gamma_i) = \text{true}$.

A predicate, $\iota$, such that $t(\pi_1) = \text{true}$ is called operational input predicate for this path.

A predicate, $\mu$, such that $\mu(\gamma_i) = \text{true}$ is called operational output predicate for this path.

For ease of notation, we introduce an $l$-sequence predicate $\text{isoperation}(\pi)$ for the first three bullets of the list in Def. 5.

$$\text{isoperation}(\{s_0, \ldots, s_l\}) = \Psi(s_0) \land \Psi(s_l) \land \bigwedge_{i=1}^{l-1} (\neg \Psi(s_i))$$

This predicate characterizes all paths of length $l$ beginning at an important state, ending at an important state and visiting only unimportant states in between (if any).

When creating an abstraction, we have a certain amount of freedom in choosing a set of important-state predicates, $\{\eta_{i}\}$, operational input predicates, $\{\iota_{j}\}$ and operational output predicates, $\{\mu_{k}\}$. However, in order to create a path predicate abstraction as in Def. 2 our choices need to fulfill the following requirements.

**Definition 6 [Abstraction Requirements]:**

For a given FSM, the set of important state predicates, $\{\eta_{i}\}$, the set of operational input predicates, $\{\iota_{j}\}$, and the set of operational output predicates, $\{\mu_{k}\}$, are chosen such that the following requirements are fulfilled:

1) The initial states are important states.
2) Every path beginning in an important state has a prefix which is an operational path according to Def. 5.
3) Consider the operational paths $\text{isoperation}(\pi)$ between two sets of important states given by $\eta_B$ and $\eta_E$:

   a) For every such path, $\text{isoperation}(\pi)$, there exists an operational input predicate, $\iota_j$, with $\iota_j(\pi_1) = \text{true}$, and an operational output predicate, $\mu_k$, with $\mu_k(\gamma_i) = \text{true}$.

   b) Consider an arbitrary pair of operational input predicates, $\iota_1, \iota_2$, from the set $\{\iota_j\}$, that are applicable in $\eta_B$. Let $l_1$ be the length of the sequence predicate $\iota_1$ and $l_2$ the length of $\iota_2$, with $l_1 \leq l_2$.

5) it holds that for every state $s_E$ satisfying $\eta_E(s_E) = \text{true}$ and every input sequence $\rho$ satisfying $t(\rho) = \text{true}$ there is an FSM-operational path $(\pi, \rho, \gamma)$ with a state sequence $\pi = (s_B, \ldots, s_E)$, beginning in $s_B$ and ending in some state $s_E$ satisfying $\eta_E(s_E) = \text{true}$, with the input sequence, $\rho$, and an output sequence, $\gamma$, satisfying $\mu(\gamma) = \text{true}$.

The first and the second requirement together ensure that all paths have prefix which is an operational path. The first requirement is made to simplify the formalism, note that a more flexible abstraction can be achieved by introducing special "initial operational paths" starting from an initial state and all ending in important states characterized by the same important state predicate.

Requirement 3 states that the I/O behavior is expressed completely in terms of unique input and output sequence predicates. Requirement 4 enforces the sequencability of operations: an operation must have the same I/O behavior no matter in what state of an important-state set $\eta_B$ it begins, and it must always reach the same set of important end states, $\eta_E$. This requirement corresponds to the second requirement of Def. 1 which says that an operation starting in a node of a particular color executes in the same way from any other node with the same color.

As it turns out, these requirements can be fulfilled by property checking if an ‘operational’ view is adopted when formulating properties. Intuitively, operations can be understood as register transfers over several clock cycles as they occur in processor instructions or bus transactions. This view on formulating properties is common in industry and is even supported by prover technology tailored for this purpose such as in [21], [22], [23]. Formalizing the operational view on SoC modules, as proposed in this paper, can contribute to bridging between informal specifications at high abstraction levels and the concrete RTL description. In the following, we use Interval Property Checking [21], [24] as the basis for the methodology proposed in this paper.

**B. Interval Property Checking**

An operation property or interval property $P$ is a pair $(A_i, C_i)$ where both $A_i$ (called assumption) and $C_i$ (called commitment) are $l$-sequence predicates. The property checker proves that if the assumption holds on the design (given by the $l$-path predicates $\text{ispath}(\pi)$ and $\text{isoutput}(\pi)$), the commitment does, too:

$$\left( \text{ispath}(\pi) \land \text{isoutput}(\pi, \gamma) \right)$$

$$\Rightarrow \quad (A_i(\pi, \rho) \Rightarrow C_i(\pi, \gamma))$$

Both sequence predicates $A_i$ and $C_i$ are defined over sequences of length $l$. The parameter $l$ is called the length of the property. Since the property is implicitly checked for all possible starting states $s_0$ (not just the initial state of the system), it is a safety property. The property check can be formulated as a SAT problem that searches for a path $\pi_0$ and an input sequence $\rho_1$ in the STG of the FSM where the implication does not hold. If found then these sequences constitute a counterexample of the property. It is a false
counterexample if the state $s_0$ in the path is unreachable from the initial state.

In order to rule out unreachable counterexamples in practice, it is common to add invariants to the proof problem [21]. The strengthened property check looks like this:

$$\Phi(s_0) \land ispath(\pi_l, \rho_l) \land isoutput(\pi_i, y_i) \Rightarrow (A_l(\pi_l, \rho_l) \Rightarrow C_l(\pi_l, y_i))$$

where $\Phi(s)$ is a state predicate characterizing an over-approximation of the reachable state set and $s_0$ is the starting state of the $l$-sequence $\pi_l$.

For our purpose of creating abstractions we write the properties in a special form. The assumption $A_l$ of a property $P$ is an $l$-sequence predicate of the form

$$A_l(\pi_l, \rho_l) = \eta_l(s_0) \land t(\rho_l).$$

where $s_0$ is the first state in the state sequence $\pi_l = (s_0, \ldots, s_l)$, $\eta_l(s_0)$ characterizes the important start states of the operation and $t$ characterizes the input sequences $\rho_l = (x_0, x_1, \ldots, x_{l-1})$ triggering the operation.

The commitment $C_l$ is an $l$-sequence predicate of the form

$$C_l(\pi_l, y_l) = \mu_l(y_l) \land isoperation(\pi_l) \land \eta_E(s_l)$$

where $s_l$ is the last state in the state sequence $\pi_l = (s_0, \ldots, s_l)$, $\eta_E(s_l)$ characterizes the important ending states of the operation and $\mu_l$ characterizes the operational output sequences $y_l = (y_1, y_2, \ldots, y_l)$ produced during the operation. The predicate $isoperation$ states that $s_0$ and $s_l$ are important states and all other states in the sequence $\pi_l$ are unimportant.

The property check effectively assures that the paths considered in the operation property are FSM-operational paths.

In practice, we can obtain the desired forms of Eq. 3 and 4 by following some coding conventions for writing properties. In particular, the state predicates $\eta_l$, the operational input predicates $t_l$ and the operational output predicates $\mu_k$ are formulated as macros or functions as supported by some property checking languages and commercial tools.

C. Property Language

For the industrial application, standardized property specification languages have been developed such as PSL (Property Specification Language) or SVA (SystemVerilog Assertions), that provide extended syntax in order to ease the writing of properties in practice. Properties written in these languages can be mapped to temporal logics such asCTL (Computation Tree Logic) or LTL (Linear-time Temporal Logic).

Interval properties as introduced in Sec. III-B can be mapped to a subset of LTL as described in the following.

Definition 7 [Interval LTL formula]:

An interval LTL formula is an LTL formula that is built using only the Boolean operators $\land, \lor, \neg$ and the “next-state” operator $X$. Let us define a generalized next-state operator $X'$ that denotes finite nestings of the next-state operator, i.e., if $p$ is an interval LTL formula, then $X'(p) = X(X'\ldots Xp)$ for $t > 0$ and $X'(p) = p$.

Definition 8 [Time-Normal Form]:

An interval LTL formula is in time-normal form if the generalized next-state operator $X'$ is applied only to atomic formulas.

Since for state sequences satisfying LTL formulae, $X(a \lor b) = Xa \lor Xb$ and $X(a \land b) = Xa \land Xb$ and $\neg Xa = X\neg a$, any interval LTL formula can be translated to time-normal form. It is easy to see how an interval LTL formula can be used to specify an $l$-sequence predicate: The generalized next-state operator refers to the state variables of the system at the different “time” points in the sequence.

D. Complete Interval Property Checking

Formulating and proving operation properties can be done with any existing property checker. However, a methodology is required that allows us to fulfill the requirements of Def. 6. This paper proposes to make extensions to Complete Interval Property Checking (C-IPC) for this purpose. C-IPC can guarantee that a given set of properties is complete in the sense that it fully covers the input/output behavior of the design. This criterion was first stated in [25] and was developed independently also in [26]. In this section, we re-describe this criterion and how it is checked. We will see that operation properties match well with this notion of coverage and that the completeness check becomes computationally tractable in combination with IPC [24], [25].

The basic concept is that a set of operation properties is complete if every input/output sequence of an FSM can be fully described by a sequence of operations and every operation is described by an operation property.

The properties are required to describe the output sequences according to certain determination requirements specifying the times and circumstances when specific output signals need to be determined through the design. “Determined” means that a signal value at a given time point is a function of the inputs at current and earlier time points. As an example for a determination requirement consider data on a bus that needs to be determined only when the “data valid” signal is asserted. A determination requirement for the data signal could be written as “if (datavalid = true) then determined(data)”. In general, a determination requirement is a pair $(s, \sigma_s)$ for a signal $s$ (ex.: data) and a guard $\sigma_s$ given as an $l$-sequence predicate (ex.: datavalid) characterizing the temporal conditions when the signal $s$ is to be determined.

Definition 9 [Complete Property Set]:

A property set $V = P_1, P_2, \ldots, P_n$ is complete if two arbitrary finite state machines satisfying all properties in the set are sequentially equivalent in the signals specified in the determination requirements at the time points characterized by the guards of the determination requirements.

Completeness is proven inductively from reset. We need to prove that every input sequence beginning from an initial state drives the design through a sequence of operations, each of which is described by an operation property. The idea is
that each operation begins at a state determined by the history of the system, i.e., by the previous inputs since reset. The operation itself ends in a state that is determined by the inputs applied during the operation. This is an inductive reasoning step proving that at the end of each operation we have reached a state that is fully determined by the input sequence since reset. Checking that an operation ends in a determined state amounts to checking whether the state is a function of only the starting state and the operational inputs. In C-IPC this basic idea is implemented in a collection of tests that can be checked automatically on the set of properties. It is important to note that the checks are performed solely on the property suite $V$ — the design is not taken into account. The checks reason on the Boolean signals mentioned in the properties. The user needs to specify what signals are inputs and outputs and what signals constitute the state variables that need to be determined in each operation according to the determination requirements mentioned above. Additionally, the user is required to specify a property graph $G = (V, E)$ where the nodes $V = \{P_i\}$ are the properties. Each property $P_i$ is a pair $(A_i, C_i)$ where both the assumption $A_i$ and the commitment $C_i$ are $l$-sequence predicates; $l$ is called the length of the property $P_i$. Every property $P$ has its own length $l_P$. The edges of the property graph describe the concatenation (sequencing) of operations. There is an edge $(P_j, P_i) \in E$ if the operation specified by $P_i$ can take place immediately after the operation specified by $P_j$. (This is the case if operation $P_j$ starts in the important state that is reached by operation $P_i$. Note that, in principle, the property graph could be determined automatically from the set of properties. However, in case the property suite is incomplete or incorrect, an automatic completeness checker can give useful debugging information if the user has supplied an intended property graph which, anyways, involves only a small extra effort.

There are three checks that are performed by the completeness engine on the property graph $G$: a case split test, a successor test and a determination test, all described below.

1) Case Split Test: The case split test checks that all paths between important states in the design are described by at least one property in the property suite, i.e., that all input scenarios in an important state are covered. The set of important states is given by the commitments $\{C_i\}$ of the properties $\{P_i\}$. For every important state (given by a commitment $C_P$) reached in an operation $P$ it is checked whether the disjunction of the assumptions $\{A_{Q_j}\}$ of all successor properties $Q_j$ completely covers the commitment $C_P$, i.e., for every path starting in a substate of the important state $C_P$ there exists an operation property $Q_j$ whose assumption $A_{Q_j}$ describes the path. Let $\{A_{Q_1}, A_{Q_2}, \ldots\}$ be the set of assignments of the successor properties, then the case split test checks if

$$C_P \Rightarrow \text{next}\left(\left(A_{Q_1} \lor A_{Q_2} \lor \ldots\right), l_P\right)$$

In this expression, $l_P$ is the length of property $P$. The assumption predicates $A_{Q_i}$ of the properties $Q_i$ are shifted in time to the end of property $P$ so that the first state of $A_{Q_i}$ coincides with the last state of $C_P$.

If the case split test succeeds, this means that for every possible input trace of the system there exists a chain of properties that is executed. However, this chain may not be uniquely determined. Therefore, the following successor test is performed.

2) Successor Test: The successor test checks whether the execution of an operation $Q$ is completely determined by every predecessor operation $P$. For every predecessor/successor pair $(P, Q) \in E$ it is checked whether the assumption $A_Q$ of property $Q$ depends solely on inputs and on signals determined by the predecessor $P$. This is checked in a SAT instance created in the following way. The set of signals mentioned in the properties $P$ and $Q$ are duplicated. The first set of signals is used to describe executions of an operation $P$ followed by operation $Q$. The second set describes operation $P$ followed by an operation not being $Q$. In both executions, the same input sequences are applied. Also, the state variables that are assumed to be determined are given the same values in both executions (in the time points specified by the guards of the determination requirements.) Let $A_P', C_P'$ and $A_Q'$ be the assumption and commitment of property $P$ and the assumption of property $Q$, respectively, expressed in the copied signals. Further, let $D = \bigwedge d_i$ be the set of determination requirements expressed as a conjunction of terms $d_i$. Each $d_i$ represents one requirement $(s, \sigma_i)$ expressed as $d_i = (\sigma_i \land \sigma'_i \Rightarrow s = s')$. Intuitively, this expression states that whenever the guard of signal $s$ is true in one of the executions the signal $s$ must have the same value in both executions.

The successor test checks the following implication on the SAT instance (with the same input values in each time frame):

$$A_P \land C_P \land A_{P}' \land C_{P}' \land D \land \text{next}(A_O, l_P) \Rightarrow \text{next}(A_Q', l_P)$$

If this implication does not hold, then there exists an input sequence such that operation $P$ is executed and the assumption of property $Q$ may hold or may not hold, depending on the other signals mentioned in the properties. This is the case if the assumption $A_Q$ was written such that it depends on some state variables other than inputs and variables determined by $P$. If the implication does hold, then the assumption of $A_Q$ uniquely determines for any input sequence applied after completion of operation $P$ whether operation $Q$ will follow or not (hence the name, “successor test”).

Having established that there exists a unique chain of operations for every input trace it remains to be shown that these operations determine the output signals as stated in the determination requirements.

3) Determination Test: The determination test checks whether each property $Q$ fulfills its determination requirements provided the predecessor operation $P$, in turn, fulfilled its determination requirements. The test creates a SAT instance that is satisfied if a determination requirement is violated, i.e., if a variable required to be determined by the property $Q$ is actually not a function of the variables determined by $P$ and/or of inputs during the operation $Q$. Similar to the successor test, the set of signals mentioned in the properties $P$ and $Q$ is duplicated in order to describe two executions. In both executions, $Q$ is followed by $P$ and the same input sequences are applied. Again, also, the state variables that are assumed to be determined are given the same values in both executions in the time points specified by the guards of the determination
requirements. Let $D_P$ and $D_Q$ be the determination requirements of property $P$ and $Q$, respectively. The determination test checks the following implication on the SAT instance (with the same input sequences applied in both executions):

$$A_P \land C_P \land A'_P \land C'_P \land D_P \Rightarrow next(D_Q, I_P)$$

If this implication does not hold, then there exists an input sequence and sequences of other signals mentioned in the properties such that $Q$ is executed after $P$ but signals that are supposed to be determined may have different values in different executions.

4) Reset test: The three tests explained above prove the hypothesis of an inductive proof that states the following: assume that a property $P \in V$ has uniquely determined its ending state then a property $Q \in V$ exists that uniquely determines the ending state as well as the output sequence of the operation solely from the ending state of $P$ and from the input sequence applied during the operation $Q$. The inductive proof is rooted at the reset state.

A reset property is a special property describing the behavior after reset, e.g., with no other restrictions to the assumption part of the property. The reset test checks whether reset can always be applied deterministically and whether reset fulfills all determination requirements. It is constructed similarly to the above tests, however for the reset property and without any predecessor property.

5) Completeness enables abstraction: In the context of this work, the completeness check described above provides an effective tool to check whether the set of important state predicates, operational input predicates and operational output predicates fulfill the abstraction requirements of Def. 6. The reset property is used to fulfill Requirement 1, after reset we are in a state described by an important state predicate. Note that in practice we ease this requirement and allow the reset property to describe a special reset operation, i.e., a sequence ending in a state described by an important state predicate. Requirement 2 is fulfilled because all operation properties are written strictly with the assumption in the form of Eq. 3 and the commitment in the form of Eq. 4. These forms ensure that the FSM paths considered conform to Def. 5. Moreover, the completeness check ensures that the full I/O behavior of the design is covered by operation properties, i.e., there is a property for every operational path in the FSM.

Requirement 3a is fulfilled because all sequences mentioned in the properties are described using only the sequence predicates we have chosen. The completeness of the property set implies that for every operational path there exists an operational input predicate and an operational output predicate that are true on that path.

Requirement 4 is fulfilled because each property written in the forms of Eq. 3 (assumption) and Eq. 4 (commitment) explicitly checks the requirement.

Note that requirement 3b (uniqueness of operational input and output predicates) is not proven on-the-fly by C-IPC. This requirement can, however, be checked in a straightforward way by comparing all pairs of input sequence predicates and output sequence predicates, respectively.

E. Property-based abstraction

Once we have a set of properties together with a set of important-state predicates, operational input predicates and operational output predicates that comply with Def. 6 we have effectively created an abstract FSM model. We will show later that the corresponding abstract Kripke model is a path predicate abstraction according to Def. 2 of the corresponding concrete Kripke model. Therefore, we also relate to this abstract FSM as an (FSM) path predicate abstraction.

Definition 10 [State abstraction function]:

The vector of important state predicate values $\alpha(s) := (\eta_1(s), \eta_2(s), \ldots)$ is the state abstraction function. It defines an abstract state value for every concrete state $s$.

The abstract initial state is encoded by the important ending states of the reset property.

Definition 11 [Abstract Input Alphabet]:

For every operational input predicate, $1_j$, there is a distinct, unique abstract input symbol $\hat{\xi}_j$. The abstract input alphabet, $\hat{X}$, is the set of all abstract input symbols. The predicate mapping function $\beta_{\text{in}}$ maps input predicates $1_j$ to abstract inputs $\hat{\xi}_j$.

Definition 12 [Abstract Output Alphabet]:

For every operational output predicate, $\mu_k$, there is a distinct, unique abstract output symbol $\hat{\gamma}_k$. The abstract output alphabet, $\hat{Y}$, is the set of all abstract output symbols. The predicate mapping function $\beta_{\text{out}}$ maps output predicates $\mu_k$ to abstract outputs $\hat{\gamma}_k$.

We will shortly define the transition function of the abstract finite state machine $\hat{M} = (\hat{S}, \hat{I}, \hat{X}, \hat{Y}, \hat{\delta}, \hat{\lambda})$. We first introduce a transition relation, $\hat{T} \subseteq \hat{S} \times \hat{X} \times \hat{S}$ and then show that the relation indeed defines the abstract transition function.

For readability, we introduce the following notation:

$$\text{input}(\hat{x}, \rho_l) = 1(\rho_l) : \hat{x} = \beta_{\text{in}}(1)$$

The predicate $\text{input}(\hat{x}, \rho_l)$ is true if and only if the input sequence $\rho_l$ satisfies the operational input predicate corresponding to the abstract input $\hat{x}$. It is used in the definitions of the abstract transition relation and the abstract output function.

Definition 13 [Abstract Transition Relation]:

We consider an abstraction function $\alpha$ such that the important-state predicates $\{\eta_i\}$ and the operational input predicates $\{1_j\}$ fulfill the requirements of Def. 6. The abstract transition relation $\hat{T}$ is:

$$\forall \pi_l = (s_0, s_1, \ldots, s_l), \forall \rho_l = (x_0, x_1, \ldots, x_{l-1}) :$$

$$\left(\alpha(s_0) = \hat{s}\right) \land \text{input}(\hat{x}, \rho_l) \land \text{ispath}(\pi_l, \rho_l)$$

$$\Rightarrow \text{isoperation}(\pi_l) \land (\alpha(s_l) = \hat{s}')$$

This definition states that an abstract transition from $\hat{s}$ to $\hat{s}'$ under $\hat{x}$ exists iff, in the concrete system, all operational paths beginning at states corresponding to $\hat{s}$ and triggered by messages corresponding to $\hat{x}$ make the FSM move to a state corresponding to $\hat{s}'$. 


**Lemma 1** [Abstract Transition Function]:
The abstract transition relation of Def. 13 implicitly defines a function, i.e., it holds that

\[(\hat{s}, \hat{x}, \hat{s}'_1) \in \hat{T} \wedge (\hat{s}, \hat{x}, \hat{s}'_2) \in \hat{T} \Rightarrow \hat{s}'_2 = \hat{s}'_1 = \hat{\delta}(\hat{s}, \hat{x})\]

This is the abstract transition function.

For a proof, see [27].

An operation in an FSM \( M \) produces a sequence of output symbols along an operational \( l \)-path that satisfies exactly one operational output predicate \( \mu_j \). In the abstract FSM, this corresponds to one abstract output symbol, \( y_j \). The abstract output function \( \hat{\lambda}(\hat{s}, \hat{x}) \) computes for a given abstract state \( \hat{s} \) and a given input symbol \( \hat{x} \in \hat{X} \) an abstract output symbol \( \hat{y}_j \in \hat{Y} \), corresponding to the operational output predicate \( \{\mu_j\} \) that is true on every operational \( l \)-path triggered in the corresponding operation.

For readability, again, we introduce the following notation:

\[output(\hat{s}, \gamma_i) = \mu_j(\gamma_i) : \hat{\beta}_{\text{out}}(\mu_j) = \hat{y}\]

The predicate \( output(\hat{s}, \gamma_i) \) identifies the message predicate that corresponds to the abstract output \( \hat{y} \).

**Definition 14** [Abstract Output Function]:
The abstract output function \( \hat{\lambda} \) is given by:

\[\hat{y} = \hat{\lambda}(\hat{s}, \hat{x}) \iff \forall \pi = (s_0, x_1, \ldots, s_{l-1}) : (\alpha(s_0) = \hat{s}) \land \text{input}(\hat{s}, \rho_1) \land \text{ispath}(\pi_1, \rho_1) \land \text{isoutput}(\pi_1, \gamma_j) \Rightarrow output(\hat{s}, \gamma_j)\]

According to this definition, the abstract output function produces an abstract output symbol \( \hat{y} \) representing exactly that message that is produced by the concrete FSM on every operational path beginning from a state in \( \hat{s} \) and triggered by \( \hat{x} \).

**IV. SOUNDERNESS of PROPERTY-BASED RTL ABSTRACTION**

In this section we will show that C-IPC-based RTL abstraction introduces an operational graph coloring on the state transition graph of an FSM, thus, creating a path predicate abstraction as discussed in Section II. This path predicate abstraction is sound with respect to LTL model checking.

**A. Kripke models vs. FSMs**

LTL model checking is defined for Kripke models. We will first show how the Kripke models for the concrete and abstract FSMs are defined. We will then relate the elements of the abstraction shown in Section III (important-state predicates, input predicates, output predicates, abstract state sets and alphabets) to the concrete and abstract Kripke models, respectively. We will show that these building blocks define a graph coloring that satisfies Def. 1.

Fig. 4 shows an example of a concrete FSM. The states \( a, b, c_1 \) and \( c_2 \) are important states. The other states are unimportant. States \( c_1 \) and \( c_2 \) belong to the same set of important states, defined by \( \eta_c(s) := (s = c_1) \lor (s = c_2) \). For simplicity, the output behavior of the FSM is not shown. There is one Boolean input variable. An arrow labeled ‘−’ (don’t-care) actually represents two edges, one labeled with ‘0’, the other labeled with ‘1’.

Fig. 5 shows the corresponding abstract FSM as derived from the concrete using the notions of Sec. III. There are three abstract states, \( \hat{a}, \hat{b} \) and \( \hat{c} \), corresponding to three important-state sets, \( \eta_{a}, \eta_{b} \) and \( \eta_{c} \), of the concrete FSM, respectively. There are seven abstract transitions, labeled with abstract input symbols, \( x_1, \ldots, x_7 \). Each input symbol corresponds to an operational input predicate, \( \iota_1, \ldots, \iota_7 \) that define input sequences (triggers) for seven operations. Abstract output symbols are not shown.

The example illustrates how operations, i.e., sets of operational paths in the FSM, are abstracted into single transitions in the abstract FSM. Take state \( \hat{a} \), for example. It has three transitions, corresponding to three operations triggered by the abstract inputs \( \hat{s}_1, \hat{s}_2 \) and \( \hat{s}_3 \). The corresponding concrete state \( a \) has only two outgoing transitions because there is only a single binary input variable in the concrete model. However, there are three paths to three different important states in the concrete model, corresponding to three operations. The same transition out of an important start state (here: \( a \)) may lead to different important ending states, if some next-state decisions in the FSM are made in unimportant states along the way (state \( w \) in the example). A major part of the following discussion will be concerned with this issue and with how to come up with an abstract Kripke model that has a separate abstract state for each abstract input symbol so that it corresponds to the abstract FSM. We will obtain it by transferring the decisions from unimportant states to important states through an equivalence transformation of the Kripke model.
Let us begin by defining how to obtain a Kripke model from an FSM. A Kripke model $K$ is a finite state transition structure $(S, I, R, A, L)$ with a set of states $S$, a set of initial states $I \subseteq S$, a transition relation $R \subseteq S \times S$, a set of atomic formulas $A$, and a valuation function $L : A \rightarrow 2^S$. The input and output behavior of the FSM may be modeled in the Kripke structure by including the FSM state set together with the inputs and outputs and output alphabets of into the state set of the Kripke model. Specifically, let $M = (S_M, I_M, X_M, Y_M, \delta_M, \lambda_M)$ be the FSM. Then, the Kripke model $K$ has the following state transition behavior.

- Set of states: $S \subseteq S_M \times X_M \times Y_M$:
  $$S = \{(s_M, x_M, y_M) | y_M = \lambda(s_M, x_M)\}$$

- Set of initial states: $I \subseteq I_M \times X_M \times Y_M$:
  $$I = \{(s_M, x_M, y_M) | s_M \in I_M \land y_M = \lambda(s_M, x_M)\}$$

- Transition relation: $R = \{(s_M, x_M, y_M), (s'_M, x'_M, y'_M) | s'_M = \delta(s_M, x_M, y_M) \land y'_M = \lambda(s'_M, x'_M)\}$.

Note that the deterministic behavior of an FSM due to the next-state function $\delta$ is reflected in the Kripke model by the fact that every state, $(s_M, x_M, y_M)$, has only transitions to next states with a unique FSM state component $s'_M$; however, it has transitions to all states with that FSM state component $s'_M$.

The atomic formulas $A$ and the labeling function $L$ of the Kripke need to be chosen such that the properties we would like to prove on the model can actually be formulated. For our purposes, we require that the atomic formulae allow to uniquely define the important-state predicates, $\eta$, and the input and output sequence predicates, $\iota$ and $\mu$. A typical choice for a set of atomic formula, $A$, derived from a digital circuit model, is the set of Boolean variables encoding the states, inputs and outputs of the FSM.

Fig. 6 shows the Kripke model corresponding to the FSM of Fig. 4. A few simplifications have been made to keep the drawing simple, as follows. The outputs are not shown, i.e. a Kripke state $s$ is labeled with a tuple $(s_M, x_M)$ where $s_M$ is the corresponding FSM state and $x_M$ is the input. Arrows represent sets of transitions. An arrow ending at a rounded box enclosing a set of states represents a set of transitions, with one transition ending at each state in the set. (This is used to represent, for example, the non-determinism of inputs to the FSM.) A node in the graph marked with a don’t-care box enclosing a set of states represents a set of transitions, to the FSM.) A node in the graph marked with a don’t-care used to represent, for example, the non-determinism of inputs of Fig. 4. A few simplifications have been made to keep the outputs of the FSM.

A set of states:

- $I = \{(s_M, x_M, y_M) | s_M \in I_M \land y_M = \lambda(s_M, x_M)\}$

- Transition relation: $R = \{(s_M, x_M, y_M), (s'_M, x'_M, y'_M) | s'_M = \delta(s_M, x_M, y_M) \land y'_M = \lambda(s'_M, x'_M)\}$.

Fig. 7: Example of an equivalence-preserving node duplication in a Kripke model

FSM input '-' represents a set of two Kripke states, one with input '0' and one with input '1'.

State predicates, $\eta_M(s_M)$, defined for states $s_M$ of the finite state machine are naturally applied to the FSM-state component of the corresponding Kripke state: $\eta_K(s_K) = \eta_K((s_M, x_M, y_M)) := \eta_M(s_M)$. Likewise, input sequence predicates, $\iota_M(\pi_1)$, and output sequence predicates, $\mu_M(\rho_1)$, are applied to the input components or output components of the states in a sequence, respectively. The notion of an operational path (Def. 5) is extended in a straightforward way from FSMs to Kripke models by considering sequences of Kripke states which each are triples of state, input and output values instead of a triple $(\pi_1, \rho_1, \gamma_1)$ of an FSM state sequence, input sequence and output sequence.

We now introduce an auxiliary equivalence transformation of the Kripke model as mentioned above. The transformation consists of a duplication of a Kripke state, $s$, and a redistribution of its outgoing transitions. The copy, $s^*$, "inherits" all incoming transitions of its original, $s$, as well as all atomic propositions that are true in $s$. The outgoing transitions are split up between the original and the copy. Thus, the transformation alters the state set $S$, the transition relation $R$ and the labeling function $L$, however, under the restriction that all LTL formulae remain valid. Fig. 7 shows an example of such a state duplication. Since also the atomic formulae valid in $s$ are copied to $s^*$ the new Kripke model is indistinguishable from the original one in terms of LTL properties. The transformation simply moves the decision about the successor of node $s$ taken in a run of the model from $s$ to the immediate predecessors of $s$. In the original graph shown on the left of Fig. 7, in state $s$, a decision is made whether to make a transition into $b$ or $c$. After state duplication, the decision is taken already in state $a$. However, the same state sequences are possible in terms of atomic propositions, namely $t-p-r$ and $t-p-q$.

The transformation is needed in order to be able to define a coloring function for the states, below. The initial Kripke model $K_0$ derived from the FSM is transformed through a sequence of such equivalence transformations $K_0 \rightarrow K_1 \rightarrow \ldots \rightarrow K_n$. Each transformation is a state duplication as described above. If we were to describe state duplications formally, we would extend the Kripke model derived from an FSM by adding a fourth component to the Kripke state $s_K$ that keeps track of the "version number" of the state. A Kripke state $s_K$ is then a 4-tuple $(s_M, x_M, y_M, v)$ with $v \in \{0, 1, 2, \ldots, v_{\text{max}}\}$. The labeling function, $L$, of the Kripke model does not evaluate the version component, i.e., a given atomic proposition holds in all versions of a state. Initially, all states in the Kripke model, $K_0$, have version number 0. The duplication transformation
$K_i \rightarrow K_{i+1}$ consists of the following steps:

1. Add a copy of a state $s_{K_i}$ to the set of states. The copy has the same FSM state $x_M$, input $x_M$ and output $y_M$ components as the original but a different version component.

2. For the new state, copy all ingoing transitions from the original state.

3. Remove some of the outgoing transitions of the original state and add them to the copy. (Which transitions are selected is described below.)

To keep the presentation readable, we do not develop full formal notations for the proof. For example, we omit denoting the version component, $v$, in the following.

The equivalence transformations are carried out in the following way. We search for unimportant states that are "$\eta$-decision states" in the Kripke model, i.e., state sets $\{s\} = \{(s_{M}, x_M, y_M)\}$ with the same FSM-state component $s_M$ but with outgoing transitions that reach different important state sets $\eta$ for different FSM-input components $x_M$. Assume the decision state set has $n$ outgoing transitions. For each predecessor state, $r$, of the decision state set, we create $n$ versions by making $n-1$ copies of $r$ as described above. Then we split the outgoing transitions of $r$ such that each version of $r$ has exactly one outgoing transition to a different decision state $s$. This effectively moves the decision in $s$ to the predecessor $r$. We repeat this for every predecessor of $s$.

The search for unimportant $\eta$-decision states begins at the predecessors of important states and moves backwards through the state transition graph along unimportant states until, finally, some important states are found to be predecessors of decision states and are duplicated. As a result of this sequence of transformations we have a Kripke model where from each important starting state only a single important ending state set can be reached.

As an example of the sequence of equivalence transformations, consider the Kripke model of Fig. 6. The predecessors of the important state set $\eta_b = \{(b,0), (b,1)\}$ are the unimportant states $ue = \{(ue,0),(ue,1)\}$. This is a decision state set because $(ue,1)$ transitions into the important state set $\eta_b$ while $(ue,0)$ transitions into $uf$, eventually reaching $\eta_e$. The predecessor state set of $ue$ is state set $ud$. For each predecessor, we make a copy, i.e., we create a second state set, $ud^*$, and move all transitions going to $(ue,1)$ to $ud^*$. The outgoing transitions to $(ue,0)$ remain with $ud$. Fig. 8 shows the resulting Kripke model.

Now, state set $ud$ has become an $\eta$-decision state set. The predecessor of this state set is $(a,1)$. Again, we make a copy, $(a,1)^*$ of this state. All transitions to $(ud, -)^*$ are moved to the copy $(a,1)^*$ while all transitions to $(ud, -)$ remain with the original $(a,1)$. The result of this transformation is depicted in Fig. 9. There are no more predecessors of decision states that need to be duplicated.

The final Kripke model is equivalent with the original Kripke model in terms of LTL properties and it has the property that from every single important starting state, every operational path starting in that state leads to only a single important ending state set. In other words, every important state can be mapped to a unique operation – it "carries" all information about input decisions along the operational path. Hence, it can be mapped to an operational input predicate, $\eta_1$, that is true on the operational path and to an output predicate, $\mu_1$, that is true on the operational path. Remember that there must exist an operational input predicate and an operational output predicate for each operational path and these predicates must be unique, according to Def. 6. Therefore, the mappings from important Kripke states to operational input predicates and operational output predicates are functions. We use them to define a coloring of the Kripke model.

**B. Operational Kripke Model Coloring**

**Definition 15 [Kripke Model Coloring Function]:**
Let $W \subseteq S$ be a subset of states called important Kripke states with $W = \{(s_{M}, x_M, y_M) | \Psi(s_M) = true \}$.

We define a surjective coloring function $c : W \mapsto \hat{W}$ with $c(s) = (\eta_1, \mu)$ where $s = (s_M, x_M, y_M)$ and

- $\eta(s_M) = true$,
- $\mu(\gamma) = true$ for every operational path starting in $s$.

In our example, the important states in the Kripke model of Fig. 9 could be colored as follows (output predicates omitted), using seven different colors:

<table>
<thead>
<tr>
<th>$(a, 0)$</th>
<th>$(a, 1)$</th>
<th>$(a, 1)^*$</th>
<th>$(b, 0)$</th>
<th>$(b, 1)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(\eta_0, 1_1)$</td>
<td>$(\eta_1, 1_2)$</td>
<td>$(\eta_0, 1_3)$</td>
<td>$(\eta_0, 1_4)$</td>
<td>$(\eta_0, 1_5)$</td>
</tr>
</tbody>
</table>

**Fig. 8:** Concrete Kripke model after duplicating node $(ud, -)$

**Fig. 9:** Concrete Kripke model after duplicating node $(a, 1)$
The operational input predicates \( t_i \) characterize the trigger sequences of the individual operations. For example, \( t_1((x_0, x_1, x_2, x_3)) = (x_0 \land \neg x_3) \) is a sequence predicate of length 4 characterizing the set of input sequences that take the FSM from an \( a \)-state to a \( c \)-state. — Note that states \((c_1, 0)\) and \((c_2, 0)\) are marked with the same color, as are states \((c_1, 1)\) and \((c_2, 1)\).

**Lemma 2** [Operationally Colored Kripke Model]:
Function \( c \) of Def. 15 defines an operational coloring of the Kripke state transition graph according to Def. 1.

This is easy to see. Remember that the important-state predicates, \( \eta \), the operational input predicates, \( t \) and the operational output predicates, \( \mu \), were chosen to fulfill the abstraction requirements of Def. 6. An operational coloring must fulfill the two conditions of Def. 1. Condition 1 is fulfilled by requirement 1 of Def. 6: all operational paths are of finite length. Condition 2 is fulfilled by requirement 3 of Def. 6. This requirement states that if a Kripke state is marked with a color \((\eta_B, t_B, \mu_B)\) then all states marked with that color are beginning states of operations that end in the same important ending state set \( \eta_E \). This state set contains important states marked with different colors, according to the different operations that may be triggered. Additionally, by construction, the Kripke model has the property that if an important state with FSM-component \( s_M \) can be reached, then all such states with FSM-component \( s_M \) can be reached. (This property is not altered by the sequence of equivalence transformations, above.) Hence, if there exists a transition between some Kripke state colored \((\eta_E, t_E, \mu_E)\) and another Kripke state colored \((\eta_E, t_E, \mu_E)\), then for all Kripke states colored \((\eta_E, t_E, \mu_E)\) there is a transition to some Kripke state colored \((\eta_E, t_E, \mu_E)\), q.e.d.

**Definition 16** [Abstract Kripke Model – STG]:
We consider the state transition graph, \( \hat{G} = (\hat{W}, \hat{R}) \) of an abstract Kripke model where
- the set of nodes is \( \hat{W} \), the image of the coloring function of Def. 15,
- for any two states \( u, w \in W \) it is \((c(u), c(w)) \in \hat{R}\) if and only if there is an operational path \((u, \ldots, w)\) in the state transition graph \( G \) of the concrete Kripke model.

Based on the abstraction functions \( \alpha, \beta_a \) and \( \beta_o \) as defined in Sec. III we can re-label the states of the abstract Kripke model with abstract FSM state, input and output symbols as \( \hat{w} = (\hat{s}_M, \hat{x}_M, \hat{y}_M) \).

Fig. 10 shows the abstract Kripke model as a result of the operational coloring of Def. 15 and after re-labeling. In the figure, a vertex is labeled with a pair of abstract state and abstract input, while abstract outputs are, again, not shown.

By the definitions of the abstract FSM transition relation \( \hat{T} \) of Def. 13 and the abstract FSM output function \( \hat{\lambda} \) of Def. 14, the abstract Kripke model corresponds to the abstract FSM \( \hat{M} \) of Sec. III. In our example, it is easy to see that the abstract Kripke model of Fig. 10 corresponds to the abstract FSM of Fig. 5.

**Theorem 2** [Path Predicate Abstraction in Kripke STGs]:
The state transition graph of the abstract Kripke model of Def. 16 is a path predicate abstraction of the state transition graph of the concrete Kripke model.

This follows directly from Def. 2 and from the fact that the coloring of Def. 15 is an operational graph coloring according to Def. 1.

Since \( \hat{G} \) is a path predicate abstraction of \( G \), the soundness Theorem 1 for path predicate abstraction applies. This means: for any sequence of colors in \( \hat{G} \) there is an identical sequence of colors in \( G \) (ignoring uncolored, i.e., unimportant states along the path), and vice versa.

**C. Model checking**

We would like to exploit the soundness of path predicate abstraction of Theorem 1 in order to perform model checking on the abstract model and prove properties that also hold on the concrete RTL implementation. We need to define the set of atomic formulas, \( \hat{A} \), and a labeling function, \( \hat{L} : \hat{A} \rightarrow 2^{\hat{W}} \), of the abstract Kripke model, and we need to define how to map abstract LTL formulas to concrete LTL formulas.

We choose the set of atomic abstract formulas to be the union of the set of abstract FSM states, \( \hat{s}_M \), abstract input alphabet, \( \hat{X}_M \) and abstract output alphabet, \( \hat{Y}_M \), i.e., \( \hat{A} = \hat{s}_M \cup \hat{X}_M \cup \hat{Y}_M \). In each abstract Kripke state, only a single atomic abstract state formula, \( \hat{s}_M \), is true, only a single atomic abstract input formula, \( \hat{x}_M \), is true and only a single atomic abstract output formula, \( \hat{y}_M \), is true. This means that each abstract Kripke state is uniquely mapped to a triple \((\hat{s}_M, \hat{x}_M, \hat{y}_M)\). (This mapping is the operational coloring of Def. 15 after re-labeling of Def. 16.)

The set of atomic formulas of the concrete Kripke model is chosen such that important and unimportant states can be uniquely discriminated and operational input predicates and operational output predicates can be formulated as interval LTL properties in terms of the atomic concrete formulas. (An obvious choice for the atomic formulas are assertions on the Boolean state, input and output variables of the original FSM.) In the following, the notation \( \eta_i \) shall represent both, the important state predicate \( \eta_i(s) \) and a purely Boolean (LTL) formula expressed in the atomic formulas of the concrete Kripke model characterizing the same set of states as \( \eta_i(s) \). In
the same way, \( \Psi \) refers to both, the state predicate \( \Psi(x) \) as well as the disjunction \( \Psi = \eta_1 \lor \eta_2 \lor \ldots \). The notation \( \eta_i \) represents both, the operational input predicate \( \eta_i((x_0, x_1, \ldots, x_{i-1})) \), and an interval LTL formula characterizing concrete paths in the Kripke model representing the same set of input sequences. Analogously, \( \mu_i \) can be interpreted either as a sequence predicate or as an interval LTL formula characterizing finite paths in the concrete Kripke model representing the same sets of output sequences.

Table I shows the mapping between abstract LTL formulas and the corresponding LTL formulas for the concrete model. Section (1) of the table is concerned with the atomic formulas of the abstract model. An atomic “FSM-state” formula \( \hat{\delta}_{M,i} \) corresponds to LTL formula \( \eta_i \). An atomic input formula \( \hat{\delta}_{M,i} \) corresponds to the interval LTL formula \( \Psi \land \eta_i \), i.e., to the set of paths beginning at important states with input sequences satisfying the input sequence predicate \( \eta_i \). In the same way, an atomic output formula \( \hat{\gamma}_{M,i} \) corresponds to the interval LTL formula \( \Psi \land \mu_i \), i.e., to the set of paths in the concrete Kripke model with output sequences satisfying the output sequence predicate \( \mu_i \). We call the concrete formulas \( \eta_i \), \( \Psi \land \eta_i \) and \( \Psi \land \mu_i \) *elementary formulas*.

Elementary formulas characterize paths over intervals of time through path predicates (i.e., prefixes of infinite paths with length \( l \geq 0 \)), as opposed to atomic formulas that are valid in single time points and are expressed as state predicates (i.e., prefixes of infinite paths with length \( l = 0 \)). Note this extension over the soundness statements in [14], [15] Elementary formulas are abstracted into atomic formulas. Since elementary formulas characterize paths we call this abstraction a path predicate abstraction.

Section (2) of Tab. I shows the translations for the temporal operators. The formulas \( \hat{\chi} \) and \( \hat{g} \) are arbitrary LTL formulas based on the atomic formulas of the abstract model. An abstract formula of the form \( \hat{X} \hat{\chi} \) corresponds to the concrete LTL formula \( X(\neg \Psi U (\Psi \land \chi)) \). Intuitively, \( \hat{X} \hat{\chi} \) means “in the operation immediately following the current operation, \( \hat{\chi} \) holds”.

In the concrete model, this means that we consider paths that begin at an important state, have a prefix of unimportant states and end in an important state in which \( \chi \) holds. (Note: The concrete formula using the Until operator represents paths of arbitrary length. In fact, the length of the unimportant-state prefix is bounded by the length of the longest operation in the Kripke model. For a given Kripke model, the concrete formula can be strengthened by a constraint on the length of the unimportant-state prefix.)

The abstract Until operator U expresses that on the abstract paths considered, \( \hat{g} \) holds until eventually a state is reached where \( \hat{\chi} \) holds. These paths correspond to concrete paths where \( g \) holds on the important states along the path until finally, an important state is reached where \( f \) holds. Note that \( g \) is a formula that says something about sequences starting in important states — and possibly expressing behavior about the unimportant states along the path. Therefore it is sensible to evaluate \( g \) only on paths starting in important states.

The “eventually \( \hat{\chi} \)” LTL formula \( \hat{F} \hat{\chi} \) is a shorthand notation for \( true U \hat{\chi} \), leading to the translation shown in the table. The formula \( \hat{G} \hat{\chi} \) states that in all abstract states on the path, \( \hat{\chi} \) holds. In the concrete Kripke model, this is expressed by \( \Psi \Rightarrow p \); on all important (starting) states on the path the interval formula \( p \) holds.

The Boolean operators \( \neg \), \( \land \) and \( \lor \) are simply transferred from the abstract to the concrete formula as shown in section (3) of Tab. I.

**Theorem 3 [LTL Soundness]:**

Let \( \hat{\phi} \) be an abstract LTL formula of one of the forms (1), (2) or (3) of the left-hand side of Tab. I where \( \hat{\chi} \) and \( \hat{g} \) are sub-formulas of these forms. The concrete LTL formula \( p \) is obtained by recursively replacing the sub-formulas with the corresponding concrete formulas on the right-hand side of Tab. I. If and only if \( \hat{\phi} \) holds in an abstract state \( \hat{w} \) then \( p \) holds in all concrete states where \( c(s) = \hat{w} \).

The proof is based on the notion of *color sequences* (cf. Def. 3) and on soundness of path predicate abstraction (cf. Theorem 1). The important states in the concrete model are colored with the elementary formulas \( \eta_i \), \( \eta_i \) and \( \mu_i \). The coloring constitutes a path predicate abstraction (Theorem 2).

We show the correspondence of each pair of LTL formulas of Tab. I by considering the color sequences produced on the paths defined by the LTL formulas. For each pair of LTL formulas, the correspondence between abstract and concrete formula is shown by the following reasoning:

- The concrete formula characterizes a set of concrete paths.
- The subset of these paths starting at important states can be described by a set of color sequences.
- The abstract formula characterizes a set of abstract paths representing the *same* set of color sequences.

Together with Theorem 1, this means that for every abstract path described by an abstract LTL formula there is a set of corresponding paths fulfilling the concrete LTL formula and for every concrete path fulfilling the concrete LTL formula there is an abstract path fulfilling the abstract LTL formula.

**D. Proof of LTL Soundness**

**Proof:** We now show, for each pair of LTL formulas in Tab. I, the equivalence of the formulas with certain sets of color sequences, based on the semantics of LTL formulas.
Section (1), elementary LTL formulas:
Elementary LTL formulas characterize paths that begin at an important (i.e., colored) state. An elementary formula $\eta_i$ characterizes all paths whose first state’s color component satisfies $\eta_i$. The same set of paths is described by the set of color sequences whose first color has the same color component $\eta_i$. The arguments for the elementary formulas $\Psi \land \alpha$ and $\Psi \land \beta_i$ are analogous. Obviously, the abstract formulas characterize the same color sequences.

Section (3), Boolean LTL formulas:
We assume that the theorem holds for the pairs of sub-formulas $\hat{f} - f$ and $\hat{g} - g$. (The validity of this assumption is proven below.) The LTL sub-formula $f$ characterizes the same set of paths that can, equivalently, be described by the set of color sequences $(\hat{w}_0, \hat{w}_1, \ldots)$ characterized by the abstract LTL sub-formula $\hat{f}$. The formula $\neg f$ holds in an important state $s_0$ if all paths $(s_0, s_1, \ldots)$ do not fulfill $f$. The set of all such paths can be described by a set of color sequences, none of which fulfill $\hat{f}$. (If a color sequence in this set would fulfill $\hat{f}$ then, according to our assumption and to Theorem 1, a concrete path starting in $s_0$ would exist that satisfies $f$, contradicting the validity of the LTL property $\neg f$ in this state.) The Boolean operators $\land$ and $\lor$ correspond to intersection and union, respectively, on sets of paths or sets of color sequences. We leave out a detailed proof for reasons of space.

Section (2), formulas with temporal operators:
The concrete formula $X(\neg \Psi U (\Psi \land f))$ describes paths $(s_0, \ldots, s_j, \ldots)$ that start at some state $s_0$ and continue with a finite sequence of zero or more unimportant states until they visit an important state $s_j$. The suffix $(s_j, \ldots)$ fulfills the LTL formula $f$. We assume that Theorem 3 holds for the pair of sub-formulas $\hat{f}$ and $f$, i.e., sub-formula $\hat{f}$ characterizes path suffixes that can, equivalently, be described as the set of color sequences described by $\hat{f}$. (Validity of assumption is proven below.) Consider now all paths fulfilling the concrete LTL formula that start at important states $s_0$ colored with $c(s_0) = \hat{w}_0$. In every such path, no state is colored between $s_0$ and $s_j$, hence, $s_j$ is colored with the next color, $\hat{w}_1$, in the color sequence $(\hat{w}_0, \hat{w}_1, \ldots)$ produced by the path. This means that the set of paths beginning at important states and characterized by the concrete LTL formula can, equivalently, be characterized by color sequences $(\hat{w}_0, \hat{w}_1, \hat{w}_2, \ldots)$ where the color sequence suffix $(\hat{w}_1, \hat{w}_2, \ldots)$ is the set of path suffixes described by the LTL sub-formula $\hat{f}$. The set of color sequences $(\hat{w}_0, \hat{w}_1, \hat{w}_2, \ldots)$ describes exactly the set of paths in the abstract model where the suffix $(\hat{w}_1, \hat{w}_2, \ldots)$ fulfills sub-formula $\hat{f}$. This set of paths $(\hat{w}_0, \hat{w}_1, \hat{w}_2, \ldots)$ is described by the abstract LTL formula $X\hat{f}$.

The concrete formula $F(\Psi \land f)$ describes paths $(s_0, s_1, \ldots, s_j, \ldots)$ with a finite (possibly empty) prefix and a suffix starting at an important state $s_j$. The suffix $(s_j, \ldots)$ fulfills the LTL formula $f$. We assume that Theorem 3 holds for the pair of sub-formulas $\hat{f}$ and $f$, i.e., sub-formula $\hat{f}$ characterizes path suffixes that can, equivalently, be described as the set of color sequences described by $\hat{f}$. (Validity of assumption is proven below.) Every concrete path fulfilling $F(\Psi \land f)$ produces a color sequence with a finite prefix and a suffix fulfilling $\hat{f}$. Hence, the color sequences produced by the path fulfill $F\hat{f}$. The set of concrete paths fulfilling $F(\Psi \land f)$ and starting in a state $s_0$ with $c(s) = \hat{w}$ can be represented by sets of color sequences, each fulfilling the abstract formula $F\hat{f}$.

The validity of the theorem for the formula pair $G\hat{f} - G(\Psi \Rightarrow f)$ follows from the duality law $\neg F p = G\neg p$.

The sub-proof for the Until operator is similar to the above proofs and is left out for reasons of space.

— In the sub-proofs, we made the assumption that the theorem holds for sub-formula pairs $\hat{f} - f$ and $\hat{g} - g$. This assumption is proven by recursively decomposing the formulas into sub-formulas, constructing a recursion tree, until, at the leaves of the tree, pairs of an abstract atomic formula and a concrete elementary formula are reached. The validity of the theorem for these formulas does not rely on sub-formulas. The validity of the assumption follows by induction from the leaves of the recursion tree backwards to the formula at the root.

V. COMPOSITIONAL PATH PREDICATE ABSTRACTIONS
In this section, we discuss how an abstract system model composed from path predicate abstractions of RTL modules can be constructed to be a sound model with respect to certain LTL properties.

Path predicate abstraction leads to time-abstract models that contain no information on how long operations actually take. A system model composed of such time-abstract modules must therefore model all possible interleavings of operations in neighboring modules. In Sec. V-A we define this abstract system model in terms of an asynchronous composition. Note that the over-approximation of interleavings between operations of different modules is reflected by the set of allowed LTL formulas in Def. 23. It may introduce false negatives but not false positives when verifying the concrete system by model checking on the composed abstract system. False negatives appear due to the over-approximation of interleavings. We may constrain this over-approximation by ensuring that synchronizations are made explicit in the abstract system model. In Sec. V-B communication and synchronization in digital hardware is discussed in general terms. Methods for modeling such communications without breaking the formal relationship between the abstract and the concrete model is introduced in Sec. V-C. In order to allow for a more flexible way of abstracting synchronization signals into abstract symbols a small extension is made to the abstraction function of Def. V-D.

Finally, in Sec. V-E it is shown that the composed abstract system model is indeed sound with respect to certain important LTL-properties.

A. Abstract System Model
We model a concrete system of $n$ finite state machines, $M = (S, I, I, X, Y_i, \delta_i, \lambda_i)$. The machines are interconnected through a common, global, input alphabet $X = X_1 \times X_2 \times \ldots \times X_n$, i.e., every machine receives the outputs of every other machine as a possible input. In order to create consistent abstractions, in our compositional approach, operational input and output predicates are replaced by message predicates. Message predicates
are used to define output sequences of some FSM of the system that serves as input sequences in other FSMs triggering operations there. Each message predicate is syntactically defined in terms of the global input space \( X \); however, semantically it relates only to the outputs of a specific sending machine. A receiving machine may access the outputs of several sending machines at the same time by evaluating the messages it receives from every sending machine.

**Definition 17 [Message Predicate]:**
A message predicate \( \mu_j(x_0, \ldots, x_{j-1}) \) is an I/O sequence predicate characterizing sequences of symbols from the (global) input alphabet, \( X \). 

Note that, for simpler notation, we write \( \mu_j((y_0, \ldots, y_{j-1})) \) when characterizing a sequence of outputs sent by a specific machine \( M_j \) in a system. The same predicate written as \( \mu_j((x_0, \ldots, x_{j-1})) \) characterizes all global system input sequences where \( M_j \) produces the specified output sequences. In the former notation the message predicate describes outgoing messages, in the latter the same messages are considered as ingoing. In practice, such message predicates may be defined as macros in a property language, relating to logic values on the interconnect lines between communicating machines. The same macro can be used without modification for ingoing and outgoing message specification.

When an individual FSM in the system reads its inputs these values may be outputs of more than one other machine. In the context of a composed system an operational input predicate, as defined within Def. 5, is therefore expressed as such a conjunction of message predicates. In the composed system, an operational input predicate \( t(p_i) \) is a conjunction of message predicates, one from each machine \( M_j \) in a system of \( n \) machines:

\[
t(p_i) = \bigwedge_{j=1}^{n} \mu_j(p_i), \quad \text{where } \mu_j \in Q_j
\]

Note that, formally, we need to define “null” message predicates for outputs from machines in the system that are ignored by a specific FSM.

The message predicates are mapped to abstract symbols through a mapping function \( \beta_j(\mu) \) for message predicates. There is a mapping function \( \beta_j \) for every FSM \( M_j \) in the system.

**Definition 18 [Message Abstraction]:**
For every message predicate \( \mu \) there is a distinct, unique, abstract message symbol \( \hat{\vartheta} \in \hat{Y}_j \). There is a one-to-one mapping between the message predicates and the abstract message symbols in \( \hat{Y}_j \) by the mapping function: \( \beta_j : \{\mu_j\} \rightarrow \hat{Y}_j \).

Using the \( \beta_j \) we can map every operational input predicate \( t \) to a corresponding \( n \)-tuple of abstract messages, \( \hat{x} \) (and vice versa).

**Definition 19 [Abstract System Alphabet]:**
The abstract system alphabet is a set of \( n \)-tuples, \( \hat{X} = \hat{Y}_1 \times \ldots \times \hat{Y}_n \) where \( \hat{Y}_j \) is the set of abstract message symbols of the \( j \)-th sending machine according to Def. 18.

The abstract state set is the product of the individual FSM state sets. But how should the transition behavior be modeled? In the concrete system, every finite state machine, \( M_i \), corresponds to a path predicate abstraction \( \hat{M}_i \). An operation in a machine \( M_i \) may comprise a sequence of state transitions but it corresponds to a single transition in the abstract model \( \hat{M}_i \). The temporal relationship between the operations in different machines based on a common clock is lost in the abstraction. Hence, in our abstract system model the abstract FSMs communicate asynchronously with each other by exchanging messages. The unknown temporal relationship between the modules is modeled using non-determinism: While each abstract FSM \( \hat{M}_i = (\hat{S}_i, \hat{I}_i, \hat{X}, \hat{Y}_i, \hat{\delta}_i, \hat{\lambda}_i) \) is still a deterministic FSM the composed model \( \hat{M} \) has a non-deterministic transition behavior modeled by a relation \( \hat{T} \) rather than a function \( \hat{\delta}() \).

**Definition 20 [Asynchronous composition]:**
The asynchronous composition \( \hat{M} \) of \( n \) path-predicate-abstracted FSMs \( M_i \) is given by \( \hat{M} = (\hat{S}, \hat{I}, \hat{X}, \hat{Y}, \hat{T}, \hat{\lambda}) \), where:

- \( \hat{S} = \hat{S}_1 \times \hat{S}_2 \times \ldots \times \hat{S}_n \) is the set of states,
- \( \hat{I} = \hat{I}_1 \times \hat{I}_2 \times \ldots \times \hat{I}_n \) is the set of initial states,
- \( \hat{X} = \hat{Y}_1 \times \hat{Y}_2 \times \ldots \times \hat{Y}_n \) is the input alphabet where \( \hat{Y}_i \) is the set of abstract primary input messages and, for all \( i, 1 \leq i \leq n \), \( \hat{Y}_i \) is the set of messages produced by the abstract FSM \( \hat{M}_i \),
- \( \hat{Y} = \hat{Y}_1 \times \ldots \times \hat{Y}_n \) is the output alphabet where \( \hat{Y}_i \) is the set of messages produced by the abstract FSM \( \hat{M}_i \),
- \( \hat{T} \) is the transition relation:

\[
(\hat{s}_1, \ldots, \hat{s}_n), \hat{x}, (\hat{s}_1', \ldots, \hat{s}_n') \in \hat{T} \iff \exists i, 1 \leq i \leq n \text{ such that } \hat{s}_i' = \hat{\delta}_i(\hat{s}_i, \hat{x}) \text{ and } \forall j, 1 \leq j \leq n, j \neq i: \hat{s}_j' = \hat{s}_j.
\]

- \( \hat{\lambda} : \hat{S} \rightarrow \hat{Y} \) is the output function, labeling every state with the output messages produced by all sub-modules with the abstract FSM \( \hat{M}_i \),

\[
\hat{\lambda}(\hat{s}_1, \ldots, \hat{s}_n) = (\hat{\lambda}_1(\hat{s}_1), \hat{\lambda}_2(\hat{s}_2), \ldots, \hat{\lambda}_n(\hat{s}_n)).
\]

This notion of an asynchronous composition will be illustrated in an example below (Fig. 14). All possible interleavings of operations between machines are represented. This model is closely related to the asynchronous product of \( \omega \)-automata in SPIN [28]. Note that the transitions in the asynchronous composition represent single transitions of sub-modules, i.e., modules never transition simultaneously but always “one after the other”.

When checking liveness properties based on this asynchronous composition fairness constraints need to be added to make sure that every sub-FSM \( \hat{M}_i \) infinitely often makes a transition (cf. finite progress assumption in [28]). This is implemented as weak fairness in SPIN.

**B. Communication schemes in digital hardware**

Communication in digital hardware relies on a few basic principles. The communicating parties need to synchronize with each other before messages can be transmitted. Also, there needs to be an agreement on how the message is actually transferred from the sender to the receiver(s).

Let us discuss the different communication schemes that we address in this paper. A first fundamental distinction is between asynchronous communication, relying on dedicated event signaling, and synchronous communication, relying on a common clock. Another distinction is to be made between implementations of communication systems that rely on timing constraints/guarantees (implicit timing) and those that do not.
In asynchronous communication the synchronization of sender and receiver is carried out through event signaling: one or more communication partners signal their being ready for communication by asserting a synchronization signal. If only one partner sends a synchronization signal then local timing constraints must guarantee that the other is ready to communicate when the synchronization signal comes. The message is then transferred either through implicit timing, meaning that the communication partners comply to timing constraints such as latency periods or setup/hold times. Or, if no implicit timing information is used, proper transmission is signaled through a handshake.

In synchronous communication the situation is similar, however, all communication partners rely on a common clock. Before a message is sent the communication partners synchronize by asserting synchronization signals (explicit synchronization). If only one communication partner sends such a signal (unilateral synchronization) then it must be guaranteed (through local timing constraints) that the other partner(s) are ready to receive the signal and the message. Because of the common clock, in synchronous communication the actual data exchange may comprise several steps (such as the beats in a burst operation on a bus). Proper reception of the data may be signaled explicitly (e.g., to accommodate for access latencies) or it may not need to be signaled because of the implicit synchronization through the clock.

C. Modeling Communication

Based on the standard communication schemes as described in Sec. V-B we show how communication can be modeled within our methodology using explicit synchronization at the abstract level.

Since our computational model is based on an asynchronous product of the individual FSMs, it is interesting to note that a system composed at the abstract level simply by connecting abstract inputs and outputs and without any abstract synchronization mechanism is actually sound with respect to the list of allowed LTL formulas of Def. 23. However, such a simple model will usually introduce false negatives due to an over-approximation of interleavings between operations of different modules. Therefore, the synchronization mechanisms of the concrete system should be reflected in an abstract synchronization model that is created to avoid these false negatives. In the following, it is explained how to model an abstract synchronization that is sound by construction for the standard communication schemes considered here.

Fig. 11 shows an example of a four-cycle handshake between two Moore machines $M_1$ and $M_2$. The handshake is carried out using signals $s$ and $r$. The signal $s$, produced by $M_1$, is asserted only in state $S$ and de-asserted in all other states. Likewise, signal $r$, produced by $M_2$, is asserted in no state other than $R$.

Before data can actually be transferred both machines need to synchronize. Assume that $M_2$ is waiting in $P$. When $M_1$ moves from $A$ to $S$ it sends a synchronization signal $s = 1$, possibly together with some data. $M_2$ is triggered by this signal and moves into $R$. Because there are no timing guarantees machine $M_1$ needs to wait in its sending state $S$ until $M_2$ has actually received the message, moved into state $R$ and acknowledged back to $M_1$ by sending the signal $r = 1$, again possibly together with some data. Machine $M_1$ then de-asserts $s$ and waits for $M_2$ to de-assert $r$ as well. Note that $M_1$ needs to wait for $M_2$ in state $B$, otherwise a new message sent during some state sequence $(B, \ldots, A, S, B)$ could go unrecognized if machine $M_2$ remains in state $R$ during that time. The four-cycle handshake built with the signals $s$ and $r$ ensures certain reachability constraints according to the four cycles: state $P$ is not left unless state $S$ is taken, $S$ is not left while in $P$, $R$ is not left while in $S$ and $B$ is not left while in $R$.

In order to obtain a composed model of path-predicate-abstracted state machines, each of the communication schemes discussed in Sec. V-B needs to be mapped to this four-cycle handshake. This is trivial in the case of asynchronous communication without local timing guarantees. In this case a four-cycle handshake communication is present in the concrete implementation as well as in its path predicate abstraction. In the other schemes discussed above only parts of the four-cycle handshake are present in the path predicate abstraction. We then need to extend the abstract models of sender and receiver(s) with the missing elements.

As an example, consider the unilateral synchronous scheme: two machines $M_1$ and $M_2$ communicating in a synchronous system with $M_1$ sending the synchronization signal and $M_2$ receiving it. Fig. 12 shows parts of their state transition graphs.

Fig. 12: Synchronous communication with unilateral synchronization

The system relies on an implicit timing guarantee stating that machine $M_2$ is always in state $P$ whenever machine $M_1$ enters state $S$. (Such implicit timing guarantees result from the communication protocol and can usually be identified easily.) Machine $M_1$ sends the synchronization signal, $z = 1$, in state $S$ to indicate that a communication operation begins.
Machine $M_2$ waits in $P$ until the synchronization signal triggers a communication operation. The operation lasts for several clock cycles in which data can be exchanged between the machines. During this operation, both machines remain in synchrony due to the common clock while they each traverse the non-important states of the operation (indicated by the smaller circles). A message predicate $\rho()$ is used to characterize the I/O sequences exchanged in the operation.

The path predicate abstractions of $M_1$ and $M_2$ with extensions

The path predicate abstractions of $M_1$ and $M_2$ as described in Sec. III are given by the state transition graphs in Fig. 13, however without the signal $r$ and the dotted arcs. The abstract states $S$ and $P$ correspond to the important starting states of the communication operation between $M_1$ and $M_2$, the abstract states $B$ and $R$ mark its end. Comparing this with the four-cycle handshake of Fig. 11 we see that states with the same names correspond to each other. The synchronization signal $z$ serves as one of the handshake signals, namely $s$ in Fig. 11. However, for a full four-cycle handshake, the dotted/italic elements of Fig. 13 need to be added so that we soundly model the communication in an abstract system that does not rely on timing guarantees. In this example, an abstract handshake signal $r$ needs to be introduced that is asserted only in state $R$. Self loops and guard conditions are added to the state transition graphs of the abstract machines $\hat{M}_1$ and $\hat{M}_2$ as shown in Fig. 13.

When are we allowed to add these elements to the state transition graphs of the path-predicate-abstracted models? As stated above, the elements (states, transitions, guard conditions) of a four-cycle handshake produce a behavior with certain reachability constraints on the product states of the composed abstract system. We may extend path predicate abstractions to full four-cycle handshake communication if and only if the corresponding concrete system has the same reachability constraints on the involved important states as the extended abstract system. This must be shown for all communication schemes considered in our methodology.

Let us begin with the case of unilateral synchronization as shown in the above example. Referring to Fig. 11, the first reachability constraint requires that state $S$ must not be entered before $P$ is entered, (i.e., $M_2$ is always ready for $M_1$ in $P$). This reachability constraint must be guaranteed by the implicit timing constraints used in the implementation. (Note that the soundness of our model relies on the validity of this constraint; see discussion below.) The other three reachability constraints ($S$ not left while in $P$, $R$ not left while in $S$ and $B$ not left while in $R$) are fulfilled through the synchronous communication operation following state $S$. They are verified by the fact that the machines transition synchronously throughout the communication operation and that this operation is unambiguously described by the message predicate $\rho()$. This same predicate is used in the formal property proofs of the communication operations in both, the sending and the receiving machine.

The discussion of the remaining communication schemes is analogous. For the case of synchronous communication with bilateral synchronization both signals, $s$ and $r$, exist in the concrete implementation and therefore also in the path predicate abstractions, as do the self-loops in state $S$ and $P$. The extension towards a full four-cycle handshake requires only the self-loops in the communication ending states, $B$ and $R$. This is justified in the same way as for the unilateral synchronous case.

For the case of asynchronous communication we identify two cases: bilateral synchronization and unilateral synchronization with an implicit timing guarantee. The first case yields a four-cycle handshake on the concrete as well as on the abstract level, as mentioned before, and needs no extension. The second case is similar to the synchronous unilateral scheme in the following respect: Instead of having a feedback signal $r$ from machine $M_2$ to $M_1$ we have implicit timing constraints (enforced, e.g., through timer circuits or counters) that enable state transitions in machine $M_1$ only if $M_2$ is guaranteed to have moved into the corresponding communication states. In other words, the timing constraints enforce the reachability constraints of the four-cycle handshake abstraction.

The soundness of the abstract model for the considered communication schemes can also be established by the following argument. If the reachability constraints are fulfilled by the implementation then the following construction yields a concrete system which is functionally equivalent with the original design and has a path predicate abstraction with a four-cycle handshake: Assume we extend the original concrete implementation of $M_2$ with an additional output signal $r$ that is evaluated by $M_1$ as in Fig. 13. Obviously, the operations corresponding to the dotted arcs are never triggered if the implementation fulfills the set of reachability constraints. Therefore, the extended implementation which has a full four-cycle handshake communication abstraction is functionally equivalent to the original implementation.

In all standard communication schemes, as they are considered here, the extended path predicate abstractions composed in an asynchronous system with communication through four-cycle handshakes, by construction, soundly model the concrete system. It only needs to be ensured that the concrete system indeed matches with one of the described communication schemes. This can be done by a simple manual inspection or can also be automated by going through a formalized check list that examines whether the path predicate abstractions created for the individual modules match with the characteristics of the considered communication schemes. In cases where implicit timing constraints are used in the implementation the soundness of our model relies on the validity of the timing constraints. In practice, however, these timing constraints are often obvious by inspection because the respective state machines have only very few states and may be always in a
state ready for communication.

Fig. 14 shows the asynchronous composition of the machines in Fig. 11. As can bee seen from the state transition graph the four-cycle handshake between the two machines is capable of modeling a synchronous communication as, for example, in Fig. 13: The starting states of the communication operation are S and P, the ending states are B and Q. In a synchronous communication the product state BQ is reached always some time after product state SP. This is reflected in the asynchronous composition of Fig. 14: All fair paths leaving SP always reach BR. (Fairness forbids infinite cycling in SP, SR or BR.)

D. Synchronization and wait-stuttering

A basic element of communication in hardware is synchronization using special synchronization signals. SoC modules remain in a waiting state until a synchronization signal to which they are sensitive becomes asserted. In our methodology, we capture the de-asserted signal value that keeps an FSM waiting by a message predicate of length 1.

Definition 21 [Waiting Operation]:

A waiting operation is an operation of length 1 in which

- the FSM moves from an important state in the set $\eta_B(\cdot)$ to an important state in the same set $\eta_B(\cdot)$, and
- the FSM produces only waiting messages according to Def. 22.

Definition 22 [Waiting Message]:

A waiting message predicate is a message predicate of length 1 that triggers only waiting operations (in other FSMs) according to Def. 21.

A synchronization signal is a (concrete) signal with the property that its de-assertion triggers a waiting operation according to Def. 22.

In systems of communicating FSMs, the FSM driving a synchronization signal may keep the signal deasserted also during operations that are of length $l$ greater than 1. We therefore need to make a small extension to our formalism of Sec. III in order to accomodate for repetitions of messages when we are not interested in the actual number of times a message is repeated. This is needed in two cases: either the repeated message triggers waiting operations or the repeated message is not received by any machine as a trigger at all.

We use the following notation for a finite number of repetitions of a sequence predicate:

$$stutter(\mu_j, l) = \mu_j \otimes \mu_j \otimes \ldots \otimes \mu_j$$

$l$ times

We change the definition of the abstract output function by redefining the output predicate used in Def. 14. This predicate may now appear in one of the following three forms:

1) $\text{output}(\hat{s}_m, \gamma) = \mu_m(\gamma)$, where $\hat{s}_m = \beta(\mu_m)$ is an abstract output symbol mapped to some message predicate $\mu_m$,

2) $\text{output}(\hat{s}_w, \gamma) = \text{stutter}(\mu_w, l)(\gamma)$, where $\hat{s}_w = \beta(\mu_w)$ is an abstract output symbol mapped to some waiting message predicate $\mu_w$ (of length 1),

3) $\text{output}(\hat{s}_m, \gamma) = \text{stutter}(\mu_{w1}, k_1) \otimes \mu_m \otimes \text{stutter}(\mu_{w2}, k_2)(\gamma)$ where $\hat{s}_{w1} = \beta(\mu_{w1})$ is an abstract output symbol mapped to some waiting message predicate $\mu_{w1}$ (of length 1), and $\hat{s}_m = \beta(\mu_m)$ is an abstract output symbol mapped to some message predicate $\mu_m$ of length $j$, and $\hat{s}_{w2} = \beta(\mu_{w2})$ is an abstract output symbol mapped to some waiting message predicate $\mu_{w2}$ (of length 1), with $k_1 + j + k_2 = l$ and $k_1 \geq 0$, $k_2 \geq 0$.

The first form is the original form used in Sec. III. (It is actually contained in the third form as a special case but is kept here for clarity.) The second form describes a finite number of repetitions of a waiting message $\mu_w$, mapped to an abstract “wait” symbol. The third form describes the outputs of an operation by three “phases”: the first phase is a (possibly empty) stuttering of wait operations, followed by a single instance of a non-waiting message, followed by a (possibly empty) stuttering of wait operations. These three phases together are abstracted into the single abstract (non-waiting) message symbol $\hat{s}_m$.

These constructs lead to a sound abstraction of the composed system because, in the receiving machines, an arbitrary number of repetitions of waiting operations has no effect and is indistinguishable from a single occurrence or even no occurrence at all.

Using this additional concept we are now able to cover all elements of standard communication schemes between finite state machines as described in Sec. V-B:

- Synchronization signals are abstracted using stuttering of message predicates $\mu_j$ of length 1. Def. 14 of the abstract output function is based on the $\text{output}_{S_k}(\cdot)$ predicate introduced above.
- Data exchange operations of lengths $l > 1$ are abstracted using message predicates of length $l$. As elaborated in Sec. V-B, such operations must be implicitly synchronized using a common clock. Abstraction of such message predicates is based on the unmodified $\text{output}(\cdot)$ predicate introduced in Sec. III. Communication works
only if such implicit synchronization is preceded by explicit synchronization (based on signaling).

- Data exchange operations of length \( l = 1 \) may or may not be directly combined with explicit synchronization. In our formalism, they are abstracted as introduced in Sec. III.

### E. Model checking on abstract system

Let us now discuss what properties can be formulated so that their validity on the abstract model implies the validity of the corresponding property on the concrete model. The abstract system model is an asynchronous product of path-predicate abstractions. The concrete system model we consider here can either be a standard synchronous product of finite state machines if all design modules share a common clock or an asynchronous composition in the form of Def. 20 if the design modules communicate asynchronously with each other. We consider LTL model checking here. A Kripke model \( \hat{K} = (\hat{S}, \hat{I}, \hat{T}, \hat{R}, I, \hat{L}) \) is derived from an asynchronous composition \( \hat{M} \) in the following way. Let \( M = (S, I, T, F, R) \) be the asynchronous composition of \( n \) path-predicate-abstracted FSMs \( M_i, 1 \leq i \leq n \). For the Kripke model \( \hat{K} \) the set of states is \( \hat{S}_K = \hat{S} \times X \), the set of initial states is \( \hat{I}_K = \hat{I} \times X \), and the transition relation is \( \hat{R}_K \subseteq \hat{S}_K \times \hat{S}_K = \{ (\hat{\delta}, \hat{x}, \hat{\gamma}, \hat{\zeta}, \hat{\delta}', \hat{\gamma}', \hat{\zeta}') \mid (\hat{\delta}, \hat{x}, \hat{\gamma}, \hat{\zeta}) \in \hat{T} \} \).

The set of atomic formulas, \( \hat{A}_K \), is composed from subsets of atomic formulas, one for each component of the system. The subset for component \( \hat{M}_i \) comprises the state set \( \hat{S}_i \), the input alphabets \( \hat{X}_i \) and the output alphabet \( \hat{Y}_i \) of that component: \( \hat{A}_i = \hat{S}_i \cup \hat{X}_i \cup \hat{Y}_i \). The overall set of atomic formulas for the system’s Kripke model is \( \hat{A}_K = \bigcup_{i=1}^{n} \hat{A}_i \).

Theorem 3 guarantees soundness of path predicate-abstract with respect to LTL model checking for the individual components of a system. For the asynchronous composition of path-predicate abstractions soundness can be established only for a restricted set of LTL properties. This is a result of time abstraction. Some product states in the abstract model may not have corresponding products of important states in the concrete model, i.e., the set of abstract product states is a superset of the product states that have corresponding concrete product states. In the specific concrete timing of the implementation some considered important states may simply not occur simultaneously. We therefore exclude LTL formulas with Boolean sub-formulas relating to the atomic propositions of more than one module.

The allowed abstract LTL formulas are defined recursively:

**Definition 23 [Allowed LTL formulas]:**

The following are LTL formulas allowed on the Kripke model of the asynchronous composition of path predicate abstractions:

- \( F \hat{f} \) if \( \hat{f} \) is an allowed LTL formula or \( \hat{f} \) is a Boolean formula in terms of only the atomic formulas \( A_i \) of a single component \( \hat{M}_i \) of the system,
- \( G\hat{f} \) if \( \hat{f} \) is an allowed LTL formula or \( \hat{f} \) is a Boolean formula in terms of only the atomic formulas \( A_i \) of a single component \( \hat{M}_i \) of the system,
- \( \hat{f} \land \hat{g} \) and \( \hat{f} \lor \hat{g} \) and \( \neg \hat{f} \) if \( \hat{f} \) and \( \hat{g} \) are allowed LTL formulas.

**Theorem 4 [LTL Soundness of Composed Model]:**

Consider an LTL formula \( \hat{\phi} \) for the abstract model according to Def. 23. The corresponding LTL formula \( \phi \) for the concrete model is obtained by applying the translation rules of Tab. 1. If the formula \( \hat{\phi} \) holds on the abstract model then the formula \( \phi \) holds also for the concrete model.

The soundness of the asynchronous composition with respect to the LTL formulas as by Def. 23 relies on the fact that every path in the concrete system is represented by one or several paths in the abstract system. This means that if there is a counterexample for a property in the concrete model then there also exists a counterexample on the abstract model.

We state the proof for the communication scheme of the synchronous communication on the concrete level. For the other communication schemes the proof is similar.

Consider an arbitrary (possibly infinite) path \( \pi \) from the initial state in the concrete system. In every product state on the path an arbitrary number of machines may be in a respective important state. In a synchronous communication operation, the sender and the receiver begin and end the communication in synchronized start and end states, i.e., at product states that are important in both, sender and receiver. We now split up the path \( \pi \) into fragments between such synchronized states. A communication fragment is one that begins at a communication start state and that ends at a communication end state. Obviously, this fragment has a corresponding abstract path fragment, because the start and end states, by construction (cf. Sec. III and Sec. V-C), have corresponding product states in the asynchronous composition.

A non-communication fragment is a path fragment that begins at the end of some communication and that ends at the beginning of the next communication. In between these states the machines do not communicate and the specific product states occurring on the path fragment are a result of the specific speed at which each module actually runs in the implementation. The asynchronous model abstracts from concrete timing and represents all possible interleavings of operations in the two modules. In every individual module, an operation on the concrete level always corresponds to an abstract transition, as proved in [14]. Hence, for such a concrete path fragment at least one abstract path fragment exists.

Each path in the concrete and in the abstract machine consists of numerous path fragments and represents a sequence of communications, in the concrete and in the abstract model, respectively. As a result of the asynchronous composition, the sequence of communications contained in the abstract model is a superset of the sequence of communications in the concrete model. Therefore, every path in the concrete system beginning at the initial state has a representation in the abstract system. If a concrete LTL formula does not hold on a specific path leaving the initial state in the concrete model then there is an abstract path leaving the initial state where the abstract LTL formula is violated, also.

Note that a property in the abstract model may fail if the corresponding property on the concrete level only holds for a specific processing speed in between synchronization events.
This is a consequence of describing the system model as an asynchronous composition. Only system behavior that is speed independent, i.e., it does not depend on the processing speed of the individual modules, can be proved on the abstract level. We use LTL rather than CTL formulas since CTL existentially quantifies over the paths. Since not every abstract path must have a concrete counterpart, the asynchronous composition, in general, is not a sound model with respect to CTL.

VI. PRACTICAL METHODOLOGY

In this section we show how sound abstractions can be created from actual implementations using standard technical languages.

The basic steps of the abstraction flow are as follows:
1) Identify all operations of the design. (Operations are finite sequences of behavior between important control states of the design.) Write macro definitions in the used property checking language for the following objects:
   • State predicates describing the important control states between operations
   • Input sequence predicates (trigger conditions)
   • Output sequence predicates (output messages)
2) Write an operation property in the form assumption \( \rightarrow \) commitment as described in Sec. III-B. Assumption and commitment must both be expressed solely in the macros defined above. Formally verify the property.
3) Formally check the completeness of the property set as described in Sec. III-D.
4) The abstract model is obtained by regarding the macro names as abstract state, input and output symbols as in Sec. III-E.

Step 1, finding the operations of a design, is a creative step with a certain degree of freedom. In practice, the intended operations are usually obvious from the specification or from the design itself. However, the choice of operations determines the level of detail of the abstract model obtained in the end. The flow above normally involves several iterations to refine the macros and properties until a complete set of properties holding on the design has been found. Every property is individually proven. Since the complete set of properties fully specifies the expected behavior of the design, any deviation from this will be detected and can be fixed. We illustrate the abstraction flow by means of an example and show how the macros and properties relate to the theoretical notions introduced earlier.

The example design is inspired by a SONET/SDH framer circuit from Alcatel Lucent that has been subject to a comprehensive case study as will be reported in Sec. VII. For the illustrations of this section, we created a simplified version of the industrial design. For this simplified version, the VHDL source code as well as the SVA properties and the abstract model in PROMELA are available at http://www.eit.unikl.de/eis/forschung/ppa.

A. Example Design

SONET and SDH are two very similar transport protocols developed for transferring large amounts of data with minimal latency. Data is structured and sent in frames where each frame can have a complex hierarchical structure of payload and header information. For our example we only look at one such header field, the frame marker. The frame marker is a constant data string used to mark the start of a frame.

Data frames are sent at a fixed frequency, however, no explicit timing information is transferred alongside the data. Small differences in the clock speeds of sender and receiver can cause loss of synchronization. Common signal distortion may also lead to data incoherencies. In this environment of physical non-idealizations it is the task of a framer to synchronize with the incoming data stream.

For our example we consider a simplified protocol where a frame consists of 64 data words with a width of 8 bits. The frame marker is the 16-bit string 'A738' (hex) and constitutes the first and second word of every frame.

Fig. 15 is a block diagram of our system consisting of a framer and a monitor. The framer synchronizes with the incoming bit stream, signal data-in, by comparing actual position of the frame marker with its expected position. The outgoing data-word is simply a re-alignment of data-in, aligned to match the actual occurrence of the frame marker. Each word is assigned a unique word ID that identifies it within its frame. The current state of the synchronization process is output by the synchronized flag indicating whether or not outgoing data words are considered correctly synchronized.

On a positive clock edge the framer receives 8 bits of the bit stream, latched in from the bit stream by a shift register external to this design and operating at 8 times the clock speed. Note that these 8 bits are possibly unaligned to the words of the frame, i.e., they may contain 8 bits which are part of two different words.

Fig. 16 explains the main control flow of the framer as we could expect to find it documented in the specification sheet of such a module. We will refer to it as the Conceptual State Machine (CSM) of the design.

The framer is implemented in VHDL. Subtasks were identified and then realized as dedicated VHDL processes. Implementing a module in this way, as the product machine of several processes, is common practice for hardware descriptions. The following short discussion of the implementation should suffice as background when we later consider the abstraction mechanisms.

The incoming data, data-in, is buffered by a shift register.
In order to establish a PPA the edges of the CSM are formalized as operations. Proving the existence of an operation is done by writing and proving operation properties for the concrete design. In the following we show how operation properties are formulated to establish a PPA as defined in previous sections.

We used the CSM in Fig. 16 to explain the behavior of the design. The same CSM we will now be used as a starting point to create a PPA. We consider the edge from “prove synchronization” mode to “in synchrony” mode. The edge is formalized as an operation property and can be checked against the concrete implementation.

```
property proven_sync is
assume:
  at t: PROVE_SYNC;
at t + 1: MarkerIn;
prove:
  at t + 1: OutOfFrame64;
at t + 1: FramePulse64;
during [t + 1..t + 63]: not ImportantState;
at t + 64: SYNC;
end property;
```

We have used a pseudo property language for illustration purposes. Of course, the same property could also be expressed using standard property languages.

Note that although the operation is a property evaluated on the concrete design it is described in terms of abstract objects. The above operation describes an abstract transition from the abstract state PROVE_SYNC to the abstract state SYNC under the trigger condition MarkerIn, producing as abstract output symbol the pair (OutOfFrame64, FramePulse64). This can be done since the abstract objects are actually macros of the property language encapsulating its relation to the actual concrete design. The state predicates, η, input sequence predicates, ρ, and output sequence predicates, γ, of our formalism are realized through these macros. They define the semantics of the abstract objects in terms of concrete RTL signals. The name of the macro is used as abstract symbol while the content defines the corresponding predicate.

The above property holds if the concrete module satisfies the following. Starting at any state characterized by the macro PROVE_SYNC and receiving any concrete input sequence characterized by the macro MarkerIn, then, 64 clock cycles later we are in some state characterized by the macro SYNC. The macros OutOfFrame64 and FramePulse64 characterize all possible output sequences in this interval. In the property these macros are "anchored" at t+1 but since they describe sequence predicates of length 64 they do characterize the entire interval of the operation. (If this were not the case the completeness of the property set would fail.) According to our formalism we must also ensure that no other important state is visited during this interval. An explicit check, as seen above, is used to ensure this. The macro ImportantState describes all important states, Ψ. It is created as a disjunction over all abstract state macros.

```
macro PROVE_SYNC : boolean :=
  not next(prev_miss, 2) and not next(synchronized) and
  next(frame_cnt, 2)=3 and
  not next(synchronized) and
  not next(prev_miss, 2) and
  (not next(sync_bit) or not next(synchronized, 2));
end macro;
```

---

The width of this register is chosen such that any bit string containing the frame marker can be identified. One VHDL process implemented is concerned with identifying any occurrence of the frame marker within this shift register and reporting at what alignment the marker was found. Note that the frame marker can be found at 8 possible alignments since it receives 8 bits of the bit-stream every cycle. The found alignment is used to align data words from the incoming bit stream.

We also implemented a process that associates a word-ID with the outgoing data words. The frame marker itself is two words wide. These words have the word IDs 0 and 1. For each subsequent word in the frame the word ID is incremented by 1.

At last we have a process checking for synchronization hits and setting the mode of operation according to Fig. 16. A synchronization hit is a frame marker found at the expected alignment and at the expected cycle according to the protocol, i.e., 64 cycles later and at unchanged alignment.

For illustrating compositional PPA we introduce this framer as a module in the simple system shown in Fig. 15. An error reporting unit, the monitor, communicates with the framer.

The monitor calculates different error rates based on this communication. (Through a serial interface (uP_serial) towards a microprocessor some of the constants of this calculation can be changed.)

Loss of frame, lof, is set for consecutive frames out of synchrony and reset by consecutive frames in synchrony. Configuration registers, lof-set and lof-reset are used, respectively, to define the number of consecutive frames. A period is defined as the interval from one period-pulse to the next. Out of frame period, of-period is set if any frame within the last period was out of frame. Similarly, loss of frame period, lof-period, is set if lof was set within the last period.

**B. Obtaining a complete property set**

From a practical point of view, finding a PPA means to formalize a Conceptual State Machine (CSM) as an abstract system model. A first candidate for such a CSM is obtained by conceptualizing over the design as it is described in the specification sheet.

---

![Fig. 16: Conceptual State Machine of the Framer](image-url)
The macro above defines the abstract state symbol PROVE_SYNC and its corresponding important-state predicate. The predicate characterizes a set of concrete important states. The identifiers used inside the macro are the RTL identifiers from the VHDL implementation. Note that we allow previous and future values of state variables to be used in the predicate. This is a useful technical extension of our formalism that allows to succinctly capture a current-state set by implicitly taking into account the unrolled transition relation of the circuit model for resolving past and future state variable references. The same state set as characterized by PROVE_SYNC could also be described using only present-state variables; however, this would result in a much more complex macro definition.

```vhdl
macro MARKER_C : unsign := 'A738' hex; end macro;

macro InputHist : unsign := (prev(data_in, 2)&prev(data_in)&data_in); end macro;

macro MarkerIn : boolean :=
  not prev(reset_n, 0) and
  if prev(reset_n, 2)=0 then
    InputHist(15 downto 0)=MARKER_C or
    InputHist(16 downto 1)=MARKER_C or
    InputHist(22 downto 7)=MARKER_C;
  end if;
end macro;

The abstract input MarkerIn characterizes the set of concrete input sequences for which a frame marker is found. Macros MARKER_C and InputHist are used only to enhance readability.

Similarly, abstract output symbols FramePulse64 and OutOfFrame64 are created using macros. They encapsulate sequence predicates of length 64.

```vhdl
macro FramePulse64 : boolean :=
  next(framepulse, 1)=1 and
  next(framepulse, 2)=0 and
  next(framepulse, 3)=0 and
  ...
  next(framepulse, 64)=0;
end macro;

macro OutOfFrame64 : boolean :=
  next(synchronized, 2)=0 and
  next(synchronized, 3)=0 and
  ...
  next(synchronized, 64)=0;
end macro;
```

So far all state information has been encoded into macros corresponding to our important state predicates. This makes the relation to our formalism straightforward. Sometimes it may, however, be beneficial to allow additional state information to be stored as state variables. By doing so we can separate what we consider control state and what we consider data path state.

In our example, the operation property “proven_sync” as shown earlier does actually not hold on our RTL implementation. The property checker returns a counterexample where a frame marker comes at the correct cycle, but with an alignment differing from previous alignment. Correct synchronization depends on both, correct timing and correct alignment of the frame marker. Therefore we need to refine our CSM such that also the alignment is represented. We could do this by creating new abstract state macros, e.g., PROVE_SYNC_Align0, PROVE_SYNC_Align1, etc. However, a better approach is to separate this information out by adding an abstract state variable called Alignment:

```vhdl
macro Alignment : unsign :=
  next(prev_alignment, 2);
end macro;
```

We also create new abstract input symbols reflecting the position of an eventual frame marker hit. A single macro encapsulates eight abstract symbols, one for each possible alignment.

```vhdl
macro MarkerPosition : unsign :=
  if InputHist(22 downto 7)=MARKER_C then 7;
  elsif InputHist(21 downto 6)=MARKER_C then 6;
  elsif InputHist(16 downto 1)=MARKER_C then 1;
  else 0;
  end if;
end macro;
```

The refined operation property, “proven_sync”, now holds for the concrete implementation.

```vhdl
property proven_sync is
  assume:
    at t : PROVE_SYNC;
    at t : MarkerIn;
    at t : MarkerPosition = Alignment;
  prove:
    at t+1: OutOfFrame64;
    at t+1: FramePulse64;
    during [t+1, t+63]: not ImportantState;
    at t+64: Alignment = Alignment @ t;
    at t+64: SYNC;
end property;
```

Note that the abstract state is now defined both by our abstract state macros, referred to as conceptual states, and the value of Alignment. We must therefore also determine Alignment together with the conceptual ending state of the operation (at t+64). (The completeness check of Sec. III-D automatically ensures that all such state information is determined.)

Note that this is merely a technical extension that allows us to concisely describe a set of operations using a single property. For example, the above property corresponds to 8 operations in our formalism: One from PROVE_SYNC_Align0 to SYNC_Align0, another from PROVE_SYNC_Align1 to SYNC_Align1, etc.

In the same way as just described, a complete set of operation properties has been formulated, proved and checked for completeness. The abstraction shown in Fig. 17 is obtained as a result of this verification.

The concrete and the abstract model could be viewed as sequentially equivalent if the abstraction functions for inputs and outputs given through the macros, e.g., MarkerIn, MarkerPos and OutOfFrame, are included in the definition of equivalence. Even if an equivalence checker was available to handle this, finding the appropriate mappings is not an easy task and would require a similar operational design view as in the approach presented here. Instead, we propose to gradually create the abstraction by formulating and proving operation properties. This also has the added advantage that the global correctness proof is split into a number of local proofs.
We have now described the abstraction of the framer in some detail. Before we discuss the composition of our abstract system we now also quickly sum up how the monitor was abstracted.

The writing of configuration registers through the microprocessor interface was verified but will not be considered in our system. We rather consider the configuration registers to remain unchanged during system operation. The generation and one for the LOS generation and one for the OOF period. We therefore create one abstract FSM for the LOF generation and one for the OOF period and LOF period generation. Since these concrete functionalities are almost trivial, these abstractions do not differ much to their corresponding concrete FSMs. Their abstraction will be understood from the following discussion on compositionality.

C. Creating a compositional abstraction

In this section we consider the abstraction of the communication. For global input and output (input and output of the system), obviously, no such additional considerations are necessary. The bit stream is a concrete global input of the considered system. The abstraction of this global input was created to reflect how it is interpreted in the framer. From the possible possible scenarios of the incoming bit stream we have created abstract input symbols that enumerate the triggers of the control path. By characterizing the bit stream, we create two abstract input signals, MarkerIn and MarkerPosition. The values of the abstract input signals correspond to the abstract symbols of our formalism. MarkerIn is a Boolean signal that is set when a frame marker is found. MarkerPosition identifies the alignment where the frame marker is found (value range from 0 to 7).

The communication within the system requires special consideration. The definitions for compositional PPA from Sec. V lead to a set of additional criteria that need to be ensured for the soundness theorem to hold for systems of PPAs. In essence these are restrictions on the input/output abstractions ensuring a one-to-one relation between concrete communication and abstract messages.

Within our system the framer and the monitor communicate using the concrete signals synchronized and framepulse. We showed the operation “proven-sync” of the framer earlier. Output message symbols FramePulse64 and OutOfFrame64 were used to abstract these signals. These symbols correspond to sequence predicates of length 64, the entire length of the operation. We follow this procedure for all operations and obtain a total of eight output symbols, e.g., FramePulse1, not_FramePulse1, FramePulse64, etc. For every operation we now have a single symbol that uniquely represents an entire output sequence.

Let us now consider the composition of our abstract system, i.e., how we can use the same message symbols as input symbols of the monitor. The concrete monitor has a simple behavior, at a framepulse it reads in the value of synchronized and accordingly updates a counter. For compositionality the PPA of the monitor should use the eight output symbols of the framer as input symbols (obviously, the symbols also need to represent the same concrete interpretation/predicate). Creating such an abstraction may be possible but not very convenient. Considering predicates of length 64 to describe this simple behavior would be cumbersome and result in a questionable abstraction not representing a natural view on this module. Instead, let us rather change our message symbols so that they describe the concrete situations in the way the monitor interprets them. Our new symbols describe only three situations. No frame pulse is present:  

\[
\begin{align*}
\text{macro NoFramePulse : boolean :=} & \\
\text{not \( next(\text{framepulse}) \);} & \\
\text{end macro;} & \\
\end{align*}
\]

Frame pulse and inactive synchronized:  

\[
\begin{align*}
\text{macro OutOfFrame : boolean :=} & \\
\text{next(\text{framepulse}) \text{ and } next(\text{synchronized})=0;} & \\
\text{end macro;} & \\
\end{align*}
\]

Frame pulse and active synchronized:  

\[
\begin{align*}
\text{macro NotOutOfFrame : boolean :=} & \\
\text{next(\text{framepulse}) \text{ and } next(\text{synchronized})\neq1;} & \\
\text{end macro;} & \\
\end{align*}
\]

In the monitor, NoFramePulse will trigger waiting operations only. In Sec. V-D we redefined the abstract output function to allow such stuttering. We will now make use of this stuttering extension and prove the output in terms of the macros above. For “proven-sync” the changed property becomes:

\[
\text{property proven_sync is} \\
\text{assume:} & \\
\text{at t : PROVE_SYNC;} & \\
\text{at t : MarkerIn;} & \\
\text{at t : MarkerPosition = Alignment;} & \\
\text{prove:} & \\
\text{at t+1: NotOutOfFrame;} & \\
\text{during\{t+2, t+64\}: NoFramePulse;} & \\
\text{during\{t+1, t+63\}: not ImportantState;} & \\
\text{at t+64: Alignment = Alignment @ t;} & \\
\text{at t+64: SYNC;} & \\
\text{end property;} & \\
\end{align*}
\]
We use form (3) of the definition of an abstract symbol as described in Sec. V-D, i.e., the one-cycle message NotOutOfFrame followed by a 63 cycle stuttering of NoFramePulse. The abstract output value is the message symbol NotOutOfFrame.

For the soundness theorem to hold the abstract communication must also obey one of the synchronization schemes of Sec. V-B. Our concrete system modules share a common clock. The communication between the framer and the monitor is abstracted into a unilateral synchronization followed by a message transfer. The monitor waits as long as it receives the NoFramePulse symbol. The two other abstract symbols, OutOfFrame and NotOutOfFrame, represent all non-waiting messages appearing in the concrete system. All communications happening in the concrete system are described based on only these messages and match the communication scheme of unilateral synchronization as described in Sec. V-B. Therefore, the abstract synchronization by a four-cycle handshake is sound by construction.

We have created a system of Compositional PPAs from an RTL implementation. For this system model Theorem 4 on LTL soundness for compositional path predicate abstractions holds. System-level properties proven on the abstraction are valid proofs for system-level properties of the actual implementation. A PPA is an FSM that can be implemented in any standard language and can used for verification purposes based on formal methods as well as based on simulation.

VII. EXPERIMENTAL RESULTS

The proposed methodology has been evaluated by means of two case studies, both based on industrial RTL designs.

The first design is an implementation of the on-chip bus protocol Flexible Peripheral Interconnect (FPI) bus, owned by Infineon Technologies. We chose the FPI bus implementation since it is a highly optimized design realizing a complex communication protocol whose correctness of implementation can only be evaluated when considering the system as a whole.

A second case study was conducted on a SONET/SDH framer implementation from Alcatel-Lucent. The source code spans 27,000 lines of RTL code. It is highly configurable so that the module can be used in a wide range of SONET/SDH applications differing in various parameters, e.g., the data rate of the analyzed input stream. Different configurations are synthesized into circuits that differ substantially with respect to their logic functionality. High configurability is implemented using a comprehensive set of library functions and modules. With this case study we intend to demonstrate that our method copes efficiently with different configurations and that it allows to create abstract modules possessing the same re-usability as the implementation.

A. Case Study: Flexible Peripheral Interconnect (FPI) bus

The FPI bus is a modular system — every master and slave interfaces the bus using a dedicated master agent or slave agent, respectively. For our experiments, we constructed a system where two peripherals act as masters and two simple memories act as slaves. As an implementation of such a system we instantiated the FPI bus with two master agents and two slave agents.

The modules of the FPI bus are listed in Table II. For each module we created path predicate abstractions, as described in this paper, based on C-IPC. A total of 32 properties, 51 macros and 3 environment constraints were developed in 1850 lines of code (LoC). The property checker OneSpin 360 MV [23], running on an Intel Core i7 CPU 860 at 2.8 GHZ, proved all properties in 1 minute and 30 seconds. The property describing block transfer operations had the longest run time with 25 seconds. The size of each module in its concrete and path predicate-abstracted version is shown in Table II. The last column shows LoC in VHDL for the concrete model and LoC in PROMELA for the abstract SPIN model.

The effort to create a complete set of properties for SoC module verification is estimated to be around 2000 LoC per person month [23]. This matches also with our experiences in this project. Since the abstract model results from these properties based on macro coding conventions as illustrated in Section VI the additional effort for creating the abstract model is small. Coding conventions similar to the ones described are, anyways, considered “good practice” among verification engineers.

The FPI bus is a modular system where every master and slave interfaces the bus using a dedicated master agent or slave agent, respectively. For our experiments, we constructed a system where two peripherals act as masters and two simple memories act as slaves. As an implementation of such a system we instantiated the FPI bus with two master agents and two slave agents.

In the industrial implementation, MasterAgent and SlaveAgent are designed as functions of a single agent module which can be configured at system startup to act as an agent for either the master or the slave. Our abstractions of the MasterAgent and SlaveAgent, however, cover only the relevant behavior of either role. We estimate that the module configured as SlaveAgent covers about 2/3 of the inputs, outputs and state bits listed in Tab. II, while the module configured as MasterAgent covers nearly all of them. Similarly, for the Bus Control Unit (BCU) we ignored certain features like debugging mode and starvation protection, and abstracted only the behavior relevant for address decoding and bus arbitration. We estimate that about 1/3 of the BCU inputs, outputs and state bits contribute to these functions. But even taking this into account, our composed abstract system soundly models concrete system functions involving approximately 750 inputs, 750 outputs and 1300 state bits. Proving global safety and liveness properties on a concrete system of this size is clearly beyond the capacities of today’s model checking technology.

Note that separating out the considered system functionality

<table>
<thead>
<tr>
<th>Module</th>
<th>#inputs</th>
<th>#outputs</th>
<th>#state bits</th>
<th>#LoC</th>
</tr>
</thead>
<tbody>
<tr>
<td>MasterAgent</td>
<td>199/17</td>
<td>202/13</td>
<td>292/17</td>
<td>3568/91</td>
</tr>
<tr>
<td>SlaveAgent</td>
<td>199/10</td>
<td>202/5</td>
<td>292/7</td>
<td>3568/40</td>
</tr>
<tr>
<td>BCU</td>
<td>258/9</td>
<td>215/4</td>
<td>941/14</td>
<td>8966/41</td>
</tr>
</tbody>
</table>

TABLE II: Sizes of concrete/abstract system components
is practically impossible in the highly optimized concrete system. It is an advantage of our methodology that it can selectively extract functionality which is in the focus of the intended verification plan.

In our experiments, it was our goal to verify that the FPI bus conforms to its protocol in any environment, independently of the specific system into which it is integrated. Therefore, we implemented a peripheral which behaves in as general a way as possible. In order to achieve this, we exploited the non-determinism native to SPIN and implemented a device that randomly makes requests from a set containing all types of FPI bus transactions.

SPIN features a number of pre-defined checks that can be run on a PROMELA model. In particular, since we model the communication between modules using the PROMELA notion of a channel, SPIN globally proves freedom of deadlock in the composed system. In our experiments this global proof took 3 minutes of CPU time on a state-of-the-art PC using 3 GB of main memory.

Moreover, we formulated a representative LTL property of type $G(a \Rightarrow Fc)$ to verify that a requested transaction is executed correctly in the system. This and similar properties to verify the correctness of transactions can be proven in about 1 minute using 0.5 GB of main memory.

The value of proving these properties on the abstract level is not only in that they establish the correctness of the abstract model but that they imply valid proofs also for the actual implementation of the FPI bus. We are not aware of any other technology that could prove system-level properties of a similarly large system using a largely automated approach based on property checking.

B. Case Study: SONET/SDH Framer

For an introduction to the functionality of a framer, please refer to Sec. VI where a simplified framer implementation was created to demonstrate our practical methodology. For the industrial design, the top-level VHDL entity was verified by two complete property sets, thereby creating two abstract modules. The first module, called main, comprises all functionality belonging to the actual framer. The second module, called monitor, computes performance data of the framer.

The behavior described by the main module is spread over all VHDL components of the design spanning 27k LoC. The size of the module depends on the configuration parameters: The number of inputs varies between 132 and 549, the number of outputs between 88 and 280, and the number of state bits between 4054 and 47213. The large state space is due to input stream buffering. The monitor functionality is contained within a subset of the components spanning 850 LoC. The module has 20 inputs, 6 outputs, and 30 state bits.

The property suites for both modules were set up to be configurable with the configuration parameters of the design. Parameterization was confined to the macro definitions so that the top-level properties defining the abstraction were generic and parameter-independent. We were able to formally verify all configurations of the design using the same property suite. As a consequence, the constructed PPA is the same for every configuration of the actual design.

Table III compares the size of the largest design configuration with that of the abstract model.

<table>
<thead>
<tr>
<th>Module</th>
<th>#inputs</th>
<th>#outputs</th>
<th>#state bits</th>
<th>#LoC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Framer</td>
<td>549/9</td>
<td>280/7</td>
<td>47213/11</td>
<td>27k/122</td>
</tr>
<tr>
<td>Monitor</td>
<td>20/13</td>
<td>6/3</td>
<td>30/12</td>
<td>850/80</td>
</tr>
</tbody>
</table>

TABLE III: Sizes of concrete/abstract system components

Despite the fact that the abstract models are very compact, they contain the relevant information for describing their system-level behavior. Complex functionality such as re-alignment of the input stream is fully verified by the property suite creating the abstract model, but it is implementation-level behavior that is not relevant for the interaction of the module with its system environment and is therefore not included in the abstract model.

The property suites comprise a total of 29 properties based on 21 macros in a total of 954 LoC. For the intended abstract model the operations where chosen to cover a full data frame with a length between 2430 and 19440 cycles, depending on the configuration.

The total manual effort including formal verification of 27k LoC in VHDL was about six person months. Compared with our estimate of 2k LoC per person month the productivity figure here is more than doubled. The main reason for this is the sparse description of functionality within the highly generic and re-usable implementation.

For the configuration with the largest state space, the entire set of properties was proven on an Intel Core i7 CPU 860 at 2.8 GHz with 8 GB of memory in less than 2 minutes. No individual property took longer than 20 seconds to prove. Maximum memory usage was 1067 MB.

As can be seen from the design examples of this case study, different implementations can have the same abstract model. This ensures the re-usability of the abstract model for all configurations of the implementation. The possibility to choose an appropriate abstraction makes it possible to extract exactly the information needed in a system-level model. The abstract model obtained here for the SONET/SDH framer fully describes the relevant interaction of the model with the system environment. Yet, it is so small that the model by itself will not impose any complexity challenges for LTL property checking at the system level, for example, when conducting the automatic checks provided by the SPIN model checker.

VIII. CONCLUSION

This paper proposes a compositional approach to system-level verification of systems with concurrent modules. It is based on creating sound abstractions for concrete RTL implementations. The semantic gap between the system level and the RTL description is closed by leveraging the expressiveness of modern property languages such as SVA, and the power of state-of-the-art property checking technology such as C-IPC.

The paper demonstrates that the model checking paradigm of SPIN is not only useful for verifying concurrent processes in software but, in the context of the proposed methodology, can also verify complex hardware implementations. Ideally,
a front-end should be created that links transaction-level modelling in SystemC to the computational models of SPIN. This would increase the scope of the proposed approach in an industrial environment creating new and promising opportunities to integrate formal verification in ESL-based design flows.

REFERENCES


